15th International Workshop on Termination

WST 2016, September 5–7, 2016, Obergurgl, Austria

Edited by

Aart Middeldorp

René Thiemann

Contents

15th International Workshop on Termination (WST 2016).
Editors: Aart Middeldorp and René Thiemann

Tool Papers

Preface

This report contains the proceedings of the *15th International Workshop on Termination (WST 2016)*, which was held in Obergurgl during September 5–7, 2016 as part of CLA (Computational Logic in the Alps) 2016. Previous termination workshops were organized in St. Andrews (1993), La Bresse (1995), Ede (1997), Dagstuhl (1999), Utrecht (2001), Valencia (2003), Aachen (2004), Seattke (2006), Paris (2007), Leipzig (2009), Edinburgh (2010), Obergurgl (2012), Bertinoro (2013), and Vienna (2014). The termination workshops traditionally bring together, in an informal setting, researchers interested in all aspects of termination, whether this interest be practical or theoretical, primary or derived. The workshop also provides a ground for cross-fertilization of ideas from term rewriting and from the different programming language communities. The friendly atmosphere enables fruitful exchanges leading to joint research and subsequent publications.

WST 2016 received 14 submissions. After light reviewing and careful deliberations the program committee decided to accept all submissions. One submission was withdrawn after acceptance. The remaining 13 papers are contained in these proceedings.

The program included an invited presentation by *Reiner Hähnle* on *Refined Resource Analysis Based on Cost Relations*. The corresponding paper is included in these proceedings. Furthermore, these proceedings contain short descriptions of several tools that participated in the 2016 Termination and Complexity Competition. This competition ran live during the workshop and the results are available at `http://www.termination-portal.org/wiki/Termination_and_Complexity_Competition_2016`.

Several persons helped to make WST 2016 a success. We are grateful to the members of the program committee and to Martina Ingenhaeff and Benjamin Winder of the organization committee of CLA 2016 for their work. A special thanks to Johannes Waldmann who made the live run of the competition possible.

Innsbruck, September 2016 *Aart Middeldorp and René Thiemann*

15th International Workshop on Termination (WST 2016).
Editors: Aart Middeldorp and René Thiemann

Organization

WST Program Committee

Ugo Dal Lago	Bologna University	
Jörg Endrullis	VU University Amsterdam	
Yukiyoshi Kameyama	University of Tsukuba	
Salvador Lucas	Universidad Politëcnica de Valencia	
Aart Middeldorp	University of Innsbruck	(co-chair)
Thomas Ströder	RWTH Aachen	
René Thiemann	University of Innsbruck	(co-chair)
Andreas Weiermann	Ghent University	

CLA Organizing Committee

Martina Ingenhaeff	University of Innsbruck	
René Thiemann	University of Innsbruck	
Benjamin Winder	University of Innsbruck	

TermComp Organizing Committee

Johannes Waldmann	HTWK Leipzig	(StarExec)
René Thiemann	University of Innsbruck	(CPF)
Akihisa Yamada	University of Innsbruck	(TPDB)

TermComp Steering Committee

Jürgen Giesl	RWTH Aachen	
Frederic Mesnard	Universitde la Réunion	
Albert Rubio	UPC Barcelona	(chair)
René Thiemann	University of Innsbruck	
Johannes Waldmann	HTWK Leipzig	

15th International Workshop on Termination (WST 2016).
Editors: Aart Middeldorp and René Thiemann

Invited Paper

15th International Workshop on Termination (WST 2016).
Editors: Aart Middeldorp and René Thiemann

Refined Resource Analysis Based on Cost Relations

Antonio Flores-Montoya and Reiner Hähnle

TU Darmstadt, Dept. of Computer Science, `aeflores|haehnle@cs.tu-darmstadt.de`

Abstract

Resource analysis based on cost relations can be used to obtain precise bounds of imperative programs with loops and complex control flows, including multi-phase loops and amortized cost. Cost relations are an integer program representation where loops are translated into recursive definitions with constraints, similar to constraint logic programs. In a first step one computes the control-flow refinement of a given cost relation that yields a structured representation of the target program. It consists of sets of possible execution patterns called *chains*. In addition, one obtains invariants and summaries for each chain that serve as a basis for further refinement. Once the refinement process is saturated, computation of bounds is performed incrementally using a cost representation denoted *cost structure*. These permit to represent complex polynomial bounds, but they can be inferred and composed by mere linear arithmetic reasoning.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages

Keywords and phrases Resource analysis, bound analysis, cost relations, amortized cost

1 Introduction

Resource analysis has a long history. The (worst-case) resource consumption of a program is an undecidable property, so there cannot be a general and precise algorithm that works for every program. The quest has been to expand the class of programs that can be automatically analyzed and to increase the precision of the analysis, while at the same time retaining scalability.

After the early work of Wegbreit [15] there have been multiple proposals for functional [11, 14], logic [8, 12] and imperative programs [4, 7, 10, 13], see [1] for a more detailed overview. The first focus was on functional [14] and logic programs [8]. There one extracts and solves recurrence relations that represent the cost of the functions or predicates of a given program. Both, extraction and solving of recurrence relations, constitute challenging problems: In functions with multiple paths, one has to generate systems of recurrence relations or one has to abstract multiple paths into a single recurrence. As a consequence, recurrence relations are often non-deterministic. And they usually depend on multiple variables.

As an adequate representation to deal with these complications, Albert et al. [2] suggested the concept of *cost relations* and applied it to the analysis of Java (bytecode).

1.1 Cost Relations

Cost relations (CR) replace recurrence relations as the result of the extraction (abstraction) step during cost analysis. They are systems of recurrence relations, each of which has an associated set of linear constraints. Constrained recurrence relations are from now on called *cost equation* (CE).

The constraint set of a CE captures its applicability conditions and also expresses the (possibly non-deterministic) relations among different variables. Cost relations can also be seen as a specific form of constraint logic program.

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 1; pp. 1:1–1:8.

Ex1: Multi-phase loop

```
1 while(i<n){
2  if(r>0){
3     i=random(n);
4     r--;
5  }else
6     i++;
7 }
```

Ex2: Amortized cost

```
1 while(l!=[]){
2  s=Cons(head(l),s);
3  if(*)
4     s=popSome(s);
5  l=tail(l);
6 }
```

```
7 popSome(List s){
8  if(s==[] || *)
9   return s;
10 else
11  popSome(tail(s));
12 }
```

Figure 1 Simple examples with challenging features

Cost relation of Ex1	**Lines**
$1 : wh(i,n,r) = 0 \quad \{i \geq n\}$	1
$2 : wh(i,n,r) = 1 + wh(i',n,r') \ \{i < n,\ r > 0,\ 0 \leq i' \leq n,\ r' = r-1\}$	1–4
$3 : wh(i,n,r) = 1 + wh(i',n,r) \ \{i < n,\ r \leq 0,\ i' = i+1\}$	1, 2, 5, 6
Cost relation of Ex2	
$4 : whA(l,s) = 0 \quad \{l = 0\}$	1
$5 : whA(l,s) = 1 + popSome(s'',s') + whA(l',s')$ $\{l \geq 1,\ s \geq 0,\ s'' = s+1,\ l' = l-1\}$	1–6
$6 : whA(l,s) = 1 + whA(l',s') \ \{l \geq 1,\ s \geq 0,\ s' = s+1,\ l' = l-1\}$	1–3, 5, 6
$7 : popSome(s,so) = 0 \quad \{s = so\}$	7–9
$8 : popSome(s,so) = 1 + popSome(s',so) \ \{s \geq 1,\ s' = s-1\}$	7, 8, 10–12

Figure 2 Cost relations of the examples and their corresponding program lines

▶ **Example 1.** Consider the code examples in Fig. 1. The cost relations in Fig. 2 represent the number of iterations/recursive calls in these examples.

Ex1 gives rise to a single CR wh with three CEs 1–3. Ex2 yields two CRs, whA and $popSome$, with three and two CEs, respectively.

Loops (in Ex1 and Ex2) as well as recursive procedures (popSome in Ex2) are uniformly represented as recursive definitions. In Ex1, each CE corresponds to a loop path: CE 1 is the exit path, CE 2 is the path taken when the conditional's guard is true, and CE 3 is the path that traverses the else branch. Fig. 2 displays the program lines reached in each CE.

The constraint sets define the applicability conditions of each CE, for example, $i < n$, $r > 0$ in CE 2. Additionally, they define the behavior resulting from executing the path which can be non-deterministic, for example, $0 \leq i' \leq n$, $r' = r - 1$ in CE 2.

There is a further source of non-determinism. There are aspects of programs that cannot be modelled precisely with cost relations, such as accesses of arrays or reference types. We represent these phenomena with an $*$ (lines 3 and 8 of Ex2). This results in CEs whose application conditions are not mutually exclusive. This is the case, for instance, with CE 7 and CE 8. ◀

1.2 Limitations of Cost Relations

There are several approaches to solve cost relations [1, 3, 5], however, all of them currently suffer from limitations:

Multi-phase loops: Ex1 is an example of a loop with two phases. CE 2 (corresponding to the "then" path) is always executed (if at all) before CE 3 (corresponding to the "else"

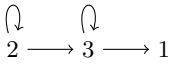
Dependency graph of wh	Chains of wh	
	Terminating	Non-terminating
	$(2)^+(3)^+1$	$(2)^+(3)^+$
	$(3)^+1$	$(3)^+$
	1	$(2)^+$

Figure 3 Dependency graph and Chains of Ex1

path). As a result, even though i can be reset r times (in line 3), the number of iterations is at most $|n| + |r|$ (where $|x|$ represents $max(x, 0)$).

Loops with resets: Consider a variant of Ex1 where line 2 is replaced with `if(r>0 && *)`. In this case CE 2 and CE 3 could interleave (CE 3 will not contain the constraint $r \leq 0$ any longer). Before i can reach n it could be reset to a value between 0 and n. This reset can happen at most r times. Therefore, an upper bound on the number of iterations would be $|n - i| + |r| + |n| * |r|$.

Amortized cost: Ex2 exemplifies amortized cost. Variable s represents a list to which we add elements of l. At some point we consume some elements of the list s with *popSome*. The functions `head`, `tail` , and `Cons` have their usual semantics.

To perform cost analysis we abstract lists to their length. Hence, within the CRs l represents the *length* of the list `l` in the program and s represents the length of the list `s`. A naive analysis would conclude that *popSome* can have at most s recursive calls, s can be at most l and *popSome* can be called at most l times in Ex2. Hence, the cost of Ex2 is proportional to at most l^2. However, it is easy to see that the cost of Ex2 is in fact linear. Alonso et al. [6] notice that cost relations cannot infer precise cost for this kind of example, because they consider only the input values of loops and function calls. Our extended notion of cost relations includes as well the output variables. For example, the variable *so* in the CRs of *popSome* represents the return value. We also developed an algorithm to compute precise amortized bounds.

All of the limitations listed above are addressed in our approach to cost analysis. In the remaining paper we sketch its main ingredients.

2 Cost Relation Control-flow Refinement

The first part of our analysis is control-flow refinement of the cost relations. For each cost relation we generate a dependency graph (call graph) whose nodes are CEs and there is an edge $e \to e'$ iff e' can be called by e. We use the CEs' constraint sets to compute the dependencies. Fig. 3 illustrates the dependency graph of Ex1.

Once we have the dependency graph, we enumerate the possible patterns of execution (chains) within that graph. Consider CEs $scc = ce_1 \cdots ce_n$ in a dependency graph that form a strongly connected component. Execution of *scc* gives rise to what we call a *phase*, where we use the notation $(ce_1 \vee \cdots \vee ce_n)^+$ to denote one or more executions of CEs in *scc* and $ce_1 \vee \cdots \vee ce_n$ to denote a single execution. The former we call an *iterative* phase, for example, $(2)^+$, the latter *non-iterative* phase, for example 1. A *chain* is a sequence of phases. The chains of Ex1 are displayed on the right part of Fig. 3. Observe that we also list the non-terminating chains (the ones that end in an iterative phase).

Next, we attempt to prove termination of each iterative phase by obtaining a lexicographical ranking function [4]. If we succeed, we can discard all the non-terminating chains ending

in that phase. In Ex1, r is a ranking function of $(2)^+$ and $n - i$ is a ranking function of $(3)^+$. Therefore, we can discard all non-terminating chains of wh.

2.1 Invariants and Summaries

A pivotal aspect of our approach is to propagate information forward and backward along each chain by polyhedral invariant computation. For instance, to compute the backward invariant of chain $(2)^+(3)^+1$, we start with the constraint set of CE 1 and apply to it the constraint set of CE 3 repeatedly until reaching a fixpoint. Then, we apply the constraint set of CE 2 to the result, again repeatedly, until reaching a fixpoint. The resulting invariant for $(2)^+(3)^+1$ is simply $i < n \wedge r > 0$ which gives us a necessary precondition for the whole chain.

In the case, where a cost relation contains output values, the backward invariant represents in particular a summary of the cost relation. Consider CR *popSome* of Ex2. The resulting chains are 7 and $(8)^+7$ (The non-terminating chain $(8)^+$ is discarded because it can be shown to be terminating). The backward invariants of 7 and $(8)^+7$ are $s = so$ and $s \geq 1 \wedge s > so$, respectively. These invariants relate the input and output values of *popSome* and thus constitute summaries of the chains' possible behaviors.

2.2 Refinement Propagation

The result of CR refinement is a set of its feasible chains with propagated summaries. We start the process with the "innermost" CRs of a program that do not make any call except to themselves (for example, *popSome* in Ex2). As a consequence, the refinement can be further propagated to the CRs that call it. Consider the situation in Ex2. We simply substitute the calls to *popSome* in CR 5 by calls to the refined chains of *popSome* (i.e. 7 and $(8)^+7$). Additionally, we enrich the constraint set of each refined CE with the summary of the called chain. Hence, CE 5 is specialized into the following two CEs (the underlined constraints are the added summaries resulting from the called chain):

$$5.1 : whA(l, s) = 1 + popSome[\mathbf{7}](s'', s') + whA(l', s')$$
$$\{l \geq 1,\ s \geq 0,\ s'' = s + 1,\ l' = l - 1,\ \underline{s'' = s'}\}$$
$$5.2 : whA(l, s) = 1 + popSome[(\mathbf{8})^+\mathbf{7}](s'', s') + whA(l', s')$$
$$\{l \geq 1,\ s \geq 0,\ s'' = s + 1,\ l' = l - 1,\ \underline{s'' > s'}\}$$

To proceed, we compute the dependency graph, feasible chains, and the summaries of the resulting CR whA. Its chains are $(5.1 \vee 5.2 \vee 6)^+4$ and 4.

3 Computing Bounds with Cost Structures

Once the refinement of all CRs is completed, we have to compute bounds. Similar as the refinements, we compute bounds incrementally, following a bottom-up approach, from the innermost to the outmost CR. Inside a single CR, we follow an incremental approach as well:

1. Compute the bound for each cost equation without considering recursive calls.
2. Compute the cost of each phase by composing the cost of their CEs.
3. Compose the cost of the phases to obtain the cost of each chain.

Therefore, the key aspect of the analysis is to represent cost bounds in such a way that they can be inferred and composed efficiently and precisely at each level (CE, phase and chain). To this end, we introduce a new data structure called *cost structure*.

3.1 Cost Structures

We represent cost with a triple $\langle E, IC, FC\rangle$ that we call *cost structure*. In a cost structure, E is a linear expression over *intermediate variables* (iv) that represents the cost. These intermediate variables are related to the variables of the CRs through two sets of constraints: *Final* FC and *non-final* IC constraints. Both constraint sets admit only constraints of a restricted form:

- *Non-final* constraints IC are expressions of the form $\sum_{k=1}^{m} iv_k \leq SE$ where SE is of the form

$$SE \quad := \quad l(\overline{iv}) \mid iv_1 * iv_2 \mid max(\overline{iv}) \mid min(\overline{iv})$$

  Here $l(\overline{iv})$ is a linear expression over intermediate variables.
- The *final* constraints FC are expressions of the form $\sum_{k=1}^{m} iv_k \leq |l(\overline{x})|$, where $l(\overline{x})$ is a linear expression over the CR variables and $|l(\overline{x})| = max(l(\overline{x}), 0)$.

This representation permits to represent complex polynomial bounds with maximum and minimum operators. At the same time, It makes it possible to define the inference and composition of cost structures in terms of simple rules and heuristics for each kind of constraint.

3.2 Computation of Bounds

Recall the three steps of computing bounds mentioned in the introduction to this section. Steps **1** and **3** involve a finite composition of cost structures. We have a specific number of cost structures and we have to compute their sum and express it in terms of different variables. For instance, to obtain the cost of $(2)^+(3)^+(1)$ we have to compose three cost structures (those of $(2)^+$, $(3)^+$ and 1) and express the result in terms of the initial variables (i, n, r) of the chain. This involves the following steps: to sum up the main cost expressions of each cost structure, to merge their non-final and final constraint sets, and to transform the final constraints so they are expressed in terms of the initial variables.

This process is based on the constraint sets of the CEs and the inferred summaries from the refinement. Final constraints are almost linear so the transformation can be implemented using Fourier-Motzkin quantifier elimination.

▶ **Example 2.** Let $\langle iv_4, \emptyset, \{iv_4 \leq |r|\}\rangle$, $\langle iv_3, \emptyset, \{iv_3 \leq |n - i|\}\rangle$, and $\langle 0, \emptyset, \emptyset\rangle$ be the cost structures of $(2)^+$, $(3)^+$, and 1, respectively. Then the composed cost structure of $(2)^+(3)^+(1)$ is $\langle iv_4 + iv_3, \emptyset, \{iv_4 \leq |r|, iv_3 \leq |n|\}\rangle$ which represents the bound $|r| + |n|$. Observe that during the phase $(2)^+$ variable i is set to an arbitrary value between 0 and n. Therefore, in the worst case, the expression $n - i$ of phase $(3)^+$ is n in the chain $(2)^+(3)^+(1)$. ◀

Step **2**, computing the cost of phases, involves the composition of an unknown number of cost structures. To realize this we generate fresh intermediate variables that represent the sums of all the instances of the previous intermediate variables. Then, we apply different heuristics to generate constraints over these new intermediate variables from the constraints of the original variables.

▶ **Example 3.** We compute the cost of $(3)^+$ in Ex1. According to the definition of CE 3, its cost (ignoring the recursive call) is 1. This can be expressed as $\langle iv_1, \emptyset, \{iv_1 \leq 1\}\rangle$. Assume the jth evaluation of CE 3 has cost $\langle iv_{1j}, \emptyset, \{iv_{1j} \leq 1\}\rangle$. Now assume that CE 3 is evaluated $\#c_3$ times in $(3)^+$. Based on that we create a new intermediate variable $iv_3 = \sum_{j=1}^{\#c_3}(iv_{1j})$

Chain/Phase/CE: Cost Structure
$(2 \vee 3)^+ : \langle iv_3 + iv_4,\ \{iv_3 \leq iv_5 + iv_6,\ iv_6 \leq iv_4 * iv_7\},\ \{iv_5 \leq |n-i|,\ iv_4 \leq |r|,\ iv_7 \leq |n|\}\rangle$
$2 : \langle iv_1,\ \emptyset,\ iv_1 \leq 1\rangle$
$3 : \langle iv_2,\ \emptyset,\ iv_2 \leq 1\rangle$
New iv definitions: $iv_3 := \sum_{j=1}^{\#c_2}(iv_{1j})$ $iv_4 := \sum_{j=1}^{\#c_3}(iv_{2j})$ $iv_6 := \sum_{j=1}^{\#c_3}(n_j)$ $iv_7 := \max_{j=1}^{\#c_3}(n_j)$

Figure 4 Cost structure of phase $(2 \vee 3)^+$ in the modified Ex1 (loop with reset) and the fresh intermediate variables defined in the process.

that represents the sum of all instances of iv_1. Now the bound of $(3)^+$ can be expressed as iv_3 and we have to generate constraints that bind iv_3 using $iv_1 \leq 1$ and the constraint set of CE 3.

One of our heuristics (called *inductive sum*) consists of applying Farkas' Lemma with a linear template $L(i, n, r)$ and the constraint set $\varphi_3 = \{i < n,\ r \leq 0,\ i' = i + 1\}$ of CE 3 to obtain a symbolic expression that satisfies: $\varphi_3 \Rightarrow (L(i, n, r) \geq 1 \wedge L(i, n, r) \geq 1 + L(i', n', r'))$. Such an expression is $(n - i)$ and it is a valid upper bound of iv_3. In this case, $n - i$ is essentially a linear ranking function of $(3)^+$. The resulting cost structure of $(3)^+$ is $\langle iv_3, \emptyset, \{iv_3 \leq |n - i|\}\rangle$ which is the one that was used in Example 2. ◀

3.3 Loop with Reset

Recall from Section 1.2 the variant of Ex1 where line 2 is replaced with `if(r>0 && *)` such that CE 2 and CE 3 can interleave. This example is interesting, because it makes use of non-final constraints to represent a non-linear bound. The main chain of the example is $(2 \vee 3)^+1$. The cost of CE 1 is 0 so let us focus on the phase $(2 \vee 3)^+$.

Fig. 4 displays the cost structure of phase $(2 \vee 3)^+$ and the fresh intermediate variables defined in the process. In these definitions $\#c_N$ represents the number of times CE N is applied. The computation proceeds incrementally. It starts with the cost structures $\langle iv_1, \emptyset, \{iv_1 \leq 1\}\rangle$ and $\langle iv_2, \emptyset, \{iv_2 \leq 1\}\rangle$ for CE 2 and 3, respectively. The variables iv_3 and iv_4 are defined in Fig. 4. The main cost expression is $iv_3 + iv_4$.

Using the inductive sum heuristics (cf. Example 3), we infer the bounds $n - i$ and r for iv_3 and iv_4, respectively. However, in contrast to the previous examples, these bounds can be influenced by other CEs in the same phase.

Expression r is unmodified in CE 3, so we can generate the constraint $iv_3 \leq |r|$. However, expression $n - i$ is reset in CE 2 to at most n (i is reset to a value between 0 and n). Hence, we add the sum of all these resets to n to obtain a bound of iv_3. We generate the constraints $iv_3 \leq iv_5 + iv_6$ and $iv_5 \leq |n - i|$ where iv_6 represents the sum of all the resets to n in CE 2.

Finally, there is no linear expression that can bind iv_6 (the sum of all n in CE 2). We apply another heuristic (*basic product*) that binds iv_6 to the product of the number of iterations of CE 2 (iv_4) and the maximum value of n along the execution (iv_7). The generated constraints are $iv_6 \leq iv_4 * iv_7$ and $iv_7 \leq |n|$ and the cost structure is now complete.

3.4 Amortized cost example

Fig. 5 contains the cost structures needed for computation of the cost of chain $(5.1 \vee 5.2 \vee 6)^+4$ of whA in Ex2 (obtained at the end of Section 2.2). Additionally, it contains the intermediate variable definitions used for the computation of the cost of phases. As before, $\#c_N$ represents

Chain/Phase/CE(Variables): Cost Structure

$[(5.1 \vee 5.2 \vee 6)^+4](l,s) : \langle iv_6 + iv_7 + iv_8 + iv_9, \emptyset, \{iv_7 + iv_8 + iv_9 \leq |l|, iv_6 \leq |l+s|\}\rangle$

- $(5.1 \vee 5.2 \vee 6)^+(l,s,l_f,s_f) : \langle iv_6 + iv_7 + iv_8 + iv_9, \emptyset, \{iv_7 + iv_8 + iv_9 \leq |l|, iv_6 \leq |l+s|\}\rangle$
  - $5.1(l,s,l',s') : \langle iv_3, \emptyset, \{iv_3 \leq 1\}\rangle$
    - $7(s,so) : \langle 0, \emptyset, \emptyset\rangle$
  - $5.2(l,s,l',s') : \langle iv_2 + iv_4, \emptyset, \{iv_4 \leq 1, iv_2 \leq |s+1-s'|\rangle$
    - $[(8)^+7](s,so) : \langle iv_2, \emptyset, \{iv_2 \leq |s|, iv_2 \leq |s-so|\}$
      - $(8)^+(s,so,s_f,so_f) : \langle iv_2, \emptyset, \{iv_2 \leq |s|, iv_2 \leq |s-s_f|\}$
        - $8(s,so,s',so') : \langle iv_1, \emptyset, \{iv_1 \leq 1\}\rangle$
      - $7(s,so) : \langle 0, \emptyset, \emptyset\rangle$
  - $6(l,s,l',s') : \langle iv_5, \emptyset, \{iv_5 \leq 1\}\rangle$
  - $4(l,s) : \langle 0, \emptyset, \emptyset\rangle$

New iv definitions:

$$iv_2 := \sum_{j=1}^{\#c_8}(iv_{1j}) \quad iv_6 := \sum_{j=1}^{\#c_{5.2}}(iv_{2j}) \quad iv_7 := \sum_{j=1}^{\#c_{5.1}}(iv_{3j}) \quad iv_8 := \sum_{j=1}^{\#c_{5.2}}(iv_{4j}) \quad iv_9 := \sum_{j=1}^{\#c_6}(iv_{5j})$$

Figure 5 Cost structures of Ex2 and intermediate variables defined in the process.

the number of times CE N is applied. The final cost structure of $(5.1 \vee 5.2 \vee 6)^+4$ represents the bound $|l| + |l+s|$ which is precise.

A key aspect in obtaining amortized cost is to consider the final values of variables. In the computation of CE 8's cost structure, the variables of the recursive call (s', so') are taken into account. In the computation of phase $(8)^+$'s cost structure, the variables of the last recursive call of the phase (s_f, so_f) are also considered. In fact iv_2 is bound by $|s - s_f|$. Intuitively, the number of recursive calls is bound by the initial value of s minus its final value s_f. In chain $(8)^+7$ the final value of s (s_f) corresponds to the return value so_f (consider $s = so$ in CE 7) and variable so is unchanged throughout phase $(8)^+$ $(so_f = so)$. Therefore, $s_f = so_f = so$ and we obtain the constraint $iv_2 \leq |s - so|$ for the chain $(8)^+7$.

Similarly, the cost structure of CE 5.2 depends on the value of the recursive call $(iv_2 \leq |s + 1 - s'|)$. Applying the *inductive sum* heuristic we can infer that $|l + s|$ is an upper bound of the sum of all the instances of $|s + 1 - s'|$ (and also of all iv_2). Let $\varphi_{5.2} = \{l \geq 1,\ s \geq 0,\ s'' = s + 1,\ l' = l - 1, s'' > s'\}$ be the constraint set of CE 5.2, we have that $\varphi_{5.2} \Rightarrow (l + s) \geq (s + 1 - s') \wedge (l + s) \geq (s + 1 - s') + (l' + s')$. The inferred constraint is $iv_6 \leq |l + s|$. We could also infer $iv_6 \leq |(l + s) - (l_f + s_f)|$ but it is not needed here.

In the cost computation of phase $(5.1 \vee 5.2 \vee 6)^+$ we have to ensure that the sums we infer are not reset or incremented in interleaving CEs. The expression $l + s$ stays invariant in CE 5.1 and 6 (l is decremented, s is incremented by 1). The expression l which bounds iv_7 is not incremented or reset in CE 5.2 or 6, but expression l also bounds iv_8 and iv_9 which allows to generate the more precise constraint $iv_7 + iv_8 + iv_9 \leq |l|$.

4 Conclusion

We showed that the limitations of existing approaches to solve cost relations listed in Section 1.2 can be overcome. This is possible by modifications to the way in which cost relations are computed, as well as to the computation of bounds. Concerning the former, we make use of control-flow refinement, an incremental, inside-out, *path-sensitive* analysis that yields stronger invariants and more precise constraints than the global analyses used hitherto. The key improvement for the computation of bounds consists in a new notion

of cost structure that permits fine-grained caching of intermediate expressions as well as the analysis of side effects by taking final values into account. Based on cost structures we defined several powerful heuristics to derive bounds. A more technical exposition of some of the ideas in this paper can be found in [9].

References

1 E. Albert, P. Arenas, S. Genaim, and G. Puebla. Closed-Form Upper Bounds in Static Cost Analysis. *Journal of Automated Reasoning*, 46(2):161–203, 2011.

2 E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of Java Bytecode. In *Proc. 16th European Symp. on Programming*, ESOP, pages 157–172. Springer, 2007.

3 E. Albert, S. Genaim, and A. N. Masud. On the Inference of Resource Usage Upper and Lower Bounds. *ACM Transactions on Computational Logic*, 14(3):22:1–22:35, 2013.

4 C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS*, volume 6337 of *LNCS*, pages 117–133. Springer, 2010.

5 D. E. Alonso-Blas, P. Arenas, and S. Genaim. Precise Cost Analysis via Local Reasoning. In D. V. Hung and M. Ogawa, editors, *Automated Technology for Verification and Analysis, 11th Intl. Symp., ATVA, Hanoi, Vietnam*, volume 8172 of *LNCS*, pages 319–333. Springer, 2013.

6 D. E. Alonso-Blas and S. Genaim. On the limits of the classical approach to cost analysis. In A. Miné and D. Schmidt, editors, *Static Analysis, 19th Intl. Symp., SAS, Deauville, France*, volume 7460 of *LNCS*, pages 405–421. Springer, Sept. 2012.

7 M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating runtime and size complexity analysis of integer programs. In *TACAS*, 2014.

8 S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

9 A. Flores Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In J. Garrigue, editor, *12th Asian Symp. on Programming Languages and Systems (APLAS)*, volume 8858 of *LNCS*, pages 275–295. Springer, Nov. 2014.

10 S. Gulwani, K. K. Mehra, and T. Chilimbi. Speed: Precise and efficient static estimation of program computational complexity. In *POPL*, pages 127–139, New York, NY, USA, 2009. ACM.

11 J. Hoffmann, K. Aehlig, and M. Hofmann. Multivariate amortized resource analysis. *SIGPLAN Not.*, 46(1):357–370, Jan. 2011.

12 A. Serrano, P. López-García, and M. V. Hermenegildo. Resource usage analysis of logic programs via abstract interpretation using sized types. *TPLP*, 14(4-5):739–754, 2014.

13 M. Sinn, F. Zuleger, and H. Veith. Difference constraints: An adequate abstraction for complexity analysis of imperative programs. In *Formal Methods in Computer-Aided Design, FMCAD, Austin, Texas, USA*, pages 144–151, 2015.

14 P. B. Vasconcelos and K. Hammond. Inferring cost equations for recursive, polymorphic and higher-order functional programs. In *Proc. 15th Intl. Conf. on Implementation of Functional Languages*, IFL, pages 86–101. Springer, 2004.

15 B. Wegbreit. Mechanical Program Analysis. *Communications of the ACM*, 18(9):528–539, September 1975.

Regular Papers

15th International Workshop on Termination (WST 2016).
Editors: Aart Middeldorp and René Thiemann

Estimation of Parallel Complexity with Rewriting Techniques*

Christophe Alias[1], Carsten Fuhs[2], and Laure Gonnord[3]

1 **INRIA & LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA), Lyon, France**
christophe.alias@ens-lyon.fr

2 **Birkbeck, University of London, United Kingdom**
carsten@dcs.bbk.ac.uk

3 **University of Lyon & LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA), Lyon, France** laure.gonnord@ens-lyon.fr

Abstract

We show how monotone interpretations – a termination analysis technique for term rewriting systems – can be used to assess the inherent parallelism of recursive programs manipulating inductive data structures. As a side effect, we show how monotone interpretations specify a parallel execution order, and how our approach extends naturally affine scheduling – a powerful analysis used in parallelising compilers – to recursive programs. This work opens new perspectives in automatic parallelisation.

1998 ACM Subject Classification E.1 Data Structures, F.2 Analysis of Algorithms and Problem Complexity F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases Complexity analysis, parallelism, monotone interpretations, termination, scheduling.

1 Introduction

The motivation of this work is the automatic transformation of sequential code into parallel code without changing its (big-step operational) semantics, only changing the computation order is allowed. We want to find out the limits of these approaches, by characterising the "maximum level of parallelism" that we can find for a given sequential implementation of an algorithm.

In this paper, we propose a way to estimate the parallel complexity which can be informally defined as the complexity of the program if it were executed on an machine with unbounded parallelism.

Such a result could come as by-product of the polyhedral-based automatic parallelisation techniques for array-based programs [6]. However, for general programs with complex data flow and inductive structures, such techniques have not been explored so far.

The contribution of this paper is a novel unifying way to express program dependencies for general programs with inductive data structures (lists, trees...) as well as a way to use complexity bounds of term rewriting systems in order to derive an estimation of this parallel complexity.

* This work was partially supported by a "BQR" funding at ENS De Lyon.

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 2; pp. 2:1–2:5.

2 Intuitions behind the notion of "parallel complexity"

```
for(i=0; i<=N; i++)
  for (j=0; j<=N; j++)
    //Block S
    {
      m1[i][j] = Integer.MIN_VALUE;
      for(k=1; k<=i; k++)
        m1[i][j] = max(m1[i][j],H[i-k][j] + W[k]);

      m2[i][j] = Integer.MIN_VALUE;
      for(k=1; k<=j; k++)
        m2[i][j] = max(m2[i][j],H[i][j-k] + W[k]);

      H[i][j] = max(0,H(i-1,j-1)+s(a[i],b[i]),
                    m1[i][j],m2[i][j]);
    }
```

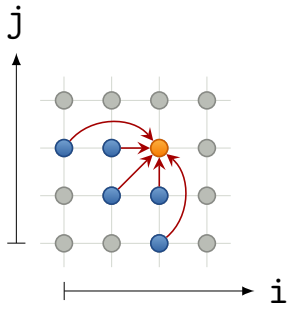


Figure 1 The Smith-Waterman sequence alignment algorithm and its dependencies. Each point (i, j) represents an execution of the block S, denoted by $\langle S, i, j \rangle$.

```
public class Tree {
  private int val;
  private Tree left;
  private Tree right;

  public int treeMax() {
    int leftMax = Integer.MIN_VALUE;
    int rightMax = Integer.MIN_VALUE;
    if (this.left != null) {
      leftMax = this.left.treeMax();  // S1
    }
    if (this.right != null) {
      rightMax = this.right.treeMax();// S2
    }
    return Math.max(this.val, Math.max(
        leftMax, rightMax));
  }
}
```

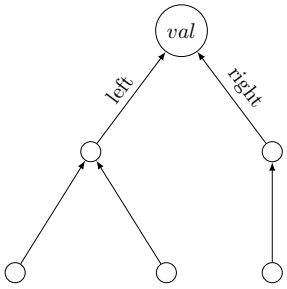


Figure 2 Maximum element of a tree algorithm and its call tree.

In this paper, we consider the derivation of a lower bound for the intrinsic parallelism of a sequential imperative program P and thus an upper bound for the complexity of a fully parallelised version of P. Figures 1 and 2 depict the two motivating examples that will be studied in the remainder of this paper. Figure 1 is a loop kernel computing the Smith-Waterman optimal sequence alignment algorithm[1]. Figure 2 is a simple recursive function to compute the maximum element of a binary tree.

Usually, the parallelism is found by analysing the data dependencies between the operations executed by the program. There is a data dependency from operation o_1 to operation o_2 in the execution of a program if o_1 is executed before o_2 and both operations access the same

[1] See `https://en.wikipedia.org/wiki/Smith-Waterman_algorithm`. We consider two sequences of the same length N.

memory location. In particular, we have a flow dependency when the result of o_1 is required by o_2. Non-flow dependencies (write after write, write after read) can be removed by playing on the memory allocation [3] and can thus be ignored. Thus, we consider only flow dependencies, referred to as dependencies in the remainder of this paper. If there is no dependency between two operations, then they may be executed in parallel. While the dependency relation is in general undecidable, in practice we can use decidable over-approximations such that a statement that two operations are independent is always correct.

In Figure 1, each point represents an execution of the block S computing `H[i][j]` for a given i and j in $[\![0, N]\!]$. Such an operation is written $\langle S, i, j\rangle$. The arrows represent the dependencies towards a given $\langle S, i, j\rangle$. For instance the diagonal arrow means that `H[i][j]` requires the value of `H[i-1][j-1]` to be computed.

In Figure 2, we depicted the execution trace of the recursive program on a tree. Here, the dependencies between computations resemble the recursive calls: the dependency graph (in the sense used for imperative programs) is the call tree.

All reorderings of computations respecting the dependencies are valid orderings (that we will name *schedule* in the following). In both cases, the dependencies we draw show a certain *potential parallelism*. Indeed, each pair of computations that do not transitively depend on each other can be executed in parallel; moreover, even a machine with infinite memory and infinitely parallel cannot do the computation in an amount of time which is shorter than the longest path in the dependency graph. The length of this longest path, referred to as *parallel complexity*, is thus an estimation of the *potential parallelism* of the program (its execution time with suitably reordered instructions on an idealised parallel machine).

Our goal in this paper is, given a sequential program P and an over-approximation of its dependency relation, to find bounds on the parallel complexity of P and the over-approximation of its dependency relation. Via the representation of the dependencies via (possibly constrained) rewrite rules, we can apply existing techniques to find such bounds for (constrained) rewrite systems (e.g., [8, 4]).

3 Computing the parallel complexity of programs

In this section we explain on the two running examples the relationship between termination and scheduling (this relationship was already explored a bit in [1]), and polynomial interpretations (more generally, proofs of complexity bounds for term rewriting) and parallel complexity. As the long-term goal is to devise compiler optimisations by automatic parallelisation for imperative programs, we consider programs with data structures such as `arrays`, `structs`, `records` and even `classes`. The main idea is that all these constructions can be classically represented as terms via the notion of *position*, and the dependencies as term rewriting rules. Contrarily to other approaches for proving termination of programs by an abstraction to rewriting (e.g., [11, 5]), we only encode the dependencies (and forget about the control flow).

3.1 First example: Smith-Waterman algorithm

For the Smith-Waterman program of Figure 1, we can derive (with polyhedral array dataflow analysis, as in [3]), forgetting about the local computations of W scores, the following dependencies as constrained rewrite rules:

$$\mathsf{dep}(i, j) \to \mathsf{dep}(i-1, j-1) : 0 \le i \le n, 0 \le j \le n$$
$$\mathsf{dep}(i, j) \to \mathsf{dep}(i-k, j) : 0 \le i \le n, 0 \le j \le n, 1 \le k \le i$$
$$\mathsf{dep}(i, j) \to \mathsf{dep}(i, j-\ell) : 0 \le i \le n, 0 \le j \le n, 1 \le \ell \le j$$

To get an estimation of the parallel complexity, following the inspiration of previous work on termination proofs for complexity analysis (e.g., [8, 1, 4]), we first try to generate a *polynomial interpretation* [10] to map function symbols and terms to polynomials:

$$Pol(\mathsf{dep}(x_1, x_2)) = x_1 + x_2$$

Essentially, this says that iteration $\langle S, x_1, x_2\rangle$ can be executed at time stamp $x_1 + x_2$. As $0 \leq x_1, x_2 \leq N$, the maximal timestamp is $2N = O(N)$. Not only does this mean that the program may be parallelised, but it provides an actual reordering of the computation, along the parallel wavefront $x_1 + x_2$.

This linear interpretation (or ranking function) provides us with a bound in $O(N^1)$ for the parallel complexity of the program. Thus, the *degree of sequentiality* is 1. As the overall runtime of the program is in $O(N^2)$, this gives us a *degree of parallelism* of $2 - 1 = 1$ [9].

Let us point out that we would have obtained the same results using affine scheduling techniques from the polyhedral model [6]. The interesting fact here is that our apparatus is not restricted to regular programs (`for` loops, arrays) as the polyhedral model. Also, current complexity analysis tools like KoAT [4] are able to compute similar results within a reasonable amount of time. The next section show how our technique applies to a recursive program on inductive data structures.

3.2 Second example: computing the maximum element in a tree

An object of class `Tree` is represented by the term $\mathsf{Tree}(val, left, right)$, for some terms val, $left$, $right$ that represent its attributes [11]. For instance, the Java object defined in Figure 3 corresponds to the following term:

$$t = \mathsf{Tree}(2, \mathsf{Tree}(3, \mathsf{null}, \mathsf{null}), \mathsf{Tree}(4, \mathsf{Tree}(7, \mathsf{null}, \mathsf{null}), \mathsf{null}))$$

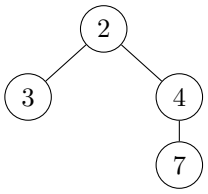


Figure 3 The tree t.

Recall that to address entries of an n-dimensional array, we use vectors $(i_1, \ldots, i_n) \in \mathbb{N}^n$ as indices or "positions". Then for an array `A`, we say that `A[`i_1`]`$\ldots$`[`i_n`]` is the "entry" at the array's position $(i_1, \ldots, i_n)$. Similarly to array entries, we would also like to address particular "entries" of a term, i.e., its *subterms*.

For more general terms, we can use a similar notion: *positions*. For our tree t, we have $Pos(t) = \{\varepsilon, 1, 2, 3, 21, 22, 23, \ldots\}$, giving an absolute way to access each element or subarray. For instance, 21 denotes the first element of the left child of the tree (i.e., the number 3).

Now let us consider the program in Figure 2. This function computes recursively the maximum value of an integer binary tree. Clearly, the computation of the maximum of a tree depends on the computation of each of its children. However, the computation of the max of each child is independent from the other. There is thus potential parallelism.

Here we observe structural dependencies from the accesses to the children of the current node. Like in the previous example, from the program we generate the following "dependency-rewrite rules" (note that similar to the dependency pair setting for termination proving [2], it suffices to consider "top-rewriting" with rewrite steps only at the root of the term):

$$\mathsf{dep}(\mathsf{Tree}(val, left, right)) \to \mathsf{dep}(left) \qquad (S1)$$
$$\mathsf{dep}(\mathsf{Tree}(val, left, right)) \to \mathsf{dep}(right) \qquad (S2)$$

We can use the following polynomial interpretation (analogous to the notion of a ranking function) to prove termination and also a complexity bound:

$$Pol(\mathsf{dep}(x_1)) = x_1 \qquad \text{and} \qquad Pol(\mathsf{Tree}(x_1, x_2, x_3)) = x_2 + x_3 + 1$$

Or, using interpretations also involving the maximum function [7]:

$$Pol(\mathsf{dep}(x_1)) = x_1 \qquad \text{and} \qquad Pol(\mathsf{Tree}(x_1, x_2, x_3)) = \max(x_2, x_3) + 1$$

Thus we interpret a tree as the maximum of its two children plus one to prove termination of the dependency relation of the original sequential program, essentially mapping a tree to its *height*. This means that the parallel complexity (i.e., the length of a chain in the dependency relation) of the program is bounded by the height of the input data structure.

Indeed, the two recursive calls could be executed in parallel, with a runtime bounded by the height of the tree on a machine with unbounded parallelism. The interpretation $Pol(\mathsf{Tree}(x_1, x_2, x_3)) = \max(x_2, x_3) + 1$ induces a wavefront for the parallel execution along the levels of the tree, i.e., the nodes at the same depth in the tree.

In contrast, the overall runtime of the original sequential program is bounded only by the *size* of the input tree, which may be exponentially larger than its depth.

4 Conclusion and Future Work

In this paper we showed some preliminary results on the (automatable) computation of the parallel complexity of programs with inductive data structures.

In the future, we will investigate a complete formalisation of these preliminary results, and test for applicability in more challenging programs like *heapsort* and *prefixsum*. As we said in the introduction, expressing the parallel complexity is the first step toward more ambitious use of rewriting techniques for program optimisation. The work in progress includes the computation of parallel schedules from the rewriting rules or their interpretation, and then parallel code generation from the obtained schedules.

References

1 Christophe Alias, Alain Darte, Paul Feautrier, and Laure Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *Proc. SAS '10*, pages 117–133, 2010.

2 Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000.

3 Denis Barthou, Albert Cohen, and Jean-François Collard. Maximal static expansion. *International Journal of Parallel Programming*, 28(3):213–243, June 2000.

4 Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Analyzing runtime and size complexity of integer programs. *ACM TOPLAS*, 2016. To appear.

5 Stephan Falke, Deepak Kapur, and Carsten Sinz. Termination analysis of C programs using compiler intermediate languages. In *Proc. RTA '11*, pages 41–50, 2011.

6 Paul Feautrier. Some efficient solutions to the affine scheduling problem, I, one-dimensional time. *International Journal of Parallel Programming*, 21(5):313–348, October 1992.

7 Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. Maximal termination. In *Proc. RTA '08*, pages 110–125, 2008.

8 Nao Hirokawa and Georg Moser. Automated complexity analysis based on the dependency pair method. In *Proc. IJCAR '08*, pages 364–379, 2008.

9 Richard M. Karp, Raymond E. Miller, and Shmuel Winograd. The organization of computations for uniform recurrence equations. *Journal of the ACM*, 14(3):563–590, 1967.

10 Dallas S. Lankford. On proving term rewriting systems are Noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA, 1979.

11 Carsten Otto, Marc Brockschmidt, Christian von Essen, and Jürgen Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *RTA '10*, pages 259–276, 2010.

Proving Termination through Conditional Termination*

Cristina Borralleras[1], Marc Brockschmidt[2], Daniel Larraz[3], Albert Oliveras[3], Enric Rodríguez-Carbonell[3], and Albert Rubio[3]

1 **Universitat de Vic**
2 **Microsoft Research, Cambridge**
3 **Universitat Politècnica de Catalunya, Barcelona**

Abstract

In this paper we use conditional termination as a technique to provide witnesses of termination of imperative programs. These witnesses are provided as a set of states for which termination is guaranteed. We show how full termination can be proved by repeatedly applying conditional termination and restricting step by step the states were non-termination may occur. In case of failure, the resulting restricted set of states can be used as starting point for a non-termination analysis. Additionally, conditional termination provides a framework for proving termination of sequences of loops and can also be applied in reachability analysis. Experiments show the effectiveness of our approach.

1998 ACM Subject Classification D.2.4

Keywords and phrases Program Analysis, Termination, Reachability, Max-SMT

1 Introduction

In its classical sense, conditional termination (e.g. [1]) aims at finding the *weakest* precondition for termination: the maximal possible set of states such that, if the execution starts at any of those states, termination is guaranteed. However, in practice the weakest precondition turns out to be too hard to compute [3]. In this paper we show that, by dropping the maximality requirement, conditions for termination can be efficiently computed, and most importantly, they can still be useful for solving a variety of problems in program analysis, among others termination and reachability.

In summary, we present the following contributions:

- A new method based on Max-SMT for finding preconditions for termination (Sect. 3).
- A framework for proving full termination of programs by repeatedly applying conditional termination and restricting step by step the states were non-termination may occur (Sect. 4). Futhermore, in case of failure, the resulting restricted set of states can be used as starting point for a non-termination analysis.
- An implementation of these techniques inside the VeryMax tool and a preliminary experimental evaluation showing the potential of our approach (Sect. 5).

2 Preliminaries

We make heavy use of the program structure and hence represent programs as graphs. For this, we fix a set of (integer) program *variables* $\mathcal{V} = \{v_1, \ldots, v_n\}$ and denote by $E(\mathcal{V})$

* This work was partially supported by the project TIN2015-69175-C4-3-R (MINECO/FEDER).

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 3; pp. 3:1–3:5.

(respectively, $F(\mathcal{V})$) the linear expressions (respectively, formulas consisting of conjunctions of linear inequalities) over the variables $\mathcal{V}$. Let $\mathcal{L}$ be the set of program *locations*, which contains a *canonical initial location* ℓ_0. Program *transitions* $\mathcal{T}$ are tuples (ℓ, τ, ℓ'), where ℓ and $\ell' \in \mathcal{L}$ represent the pre- and post-location respectively, and $\tau \in F(\mathcal{V} \cup \mathcal{V}')$ describes the transition relation. Here $\mathcal{V}' = \{v'_1, \ldots, v'_n\}$ are the *post-variables*, i.e., the values of the variables after the transition. In what follows, $\varphi' \in F(\mathcal{V}')$ is the version of φ using primed variables. More precisely, in this work we consider transition relations τ of the form $\mathit{guard}(\tau) \wedge \mathit{update}(\tau)$, where $\mathit{guard}(\tau) \in F(\mathcal{V})$ and $\mathit{update}(\tau)$ is a conjunction of linear equations of the form $v'_i = e_i$, with $e_i \in E(\mathcal{V})$.

A *program* $\mathcal{P} = (\mathcal{L}, \mathcal{T})$ is identified with its *control-flow graph* (CFG), a directed graph in which edges are the transitions $\mathcal{T}$ and nodes are the locations $\mathcal{L}$ (since transitions only have conjunctions of linear inequalities, disjunctions are expressed with several transitions). A *program component* $\mathcal{C}$ of a program $\mathcal{P}$ is the set of transitions of an SCC of the control-flow graph. Its *entry transitions* $\mathcal{E}_\mathcal{C}$ are those transitions $t = (\ell, \tau, \ell')$ such that $t \notin \mathcal{C}$ but ℓ' appears in $\mathcal{C}$ (and in this case ℓ' is called an *entry location*), while its *exit transitions* are such that $t \notin \mathcal{C}$ but ℓ appears in $\mathcal{C}$ (and then ℓ is an *exit location*).

A *state* $s = (\ell, \boldsymbol{v})$ consists of a location $\ell \in \mathcal{L}$ and a *valuation* $\boldsymbol{v} : \mathcal{V} \to \mathbb{Z}$. *Initial* states are of the form $(\ell_0, \boldsymbol{v})$. We denote an *evaluation step* with transition $t = (\ell, \tau, \ell')$ by $(\ell, \boldsymbol{v}) \to_t (\ell', \boldsymbol{v'})$, where the valuations $\boldsymbol{v}$, $\boldsymbol{v'}$ satisfy the formula τ of t. We use $\to_\mathcal{P}$ if we do not care about the executed transition, and $\to^*_\mathcal{P}$ to denote the transitive-reflexive closure of $\to_\mathcal{P}$. Sequences of evaluation steps are called *evaluations*. We say that a state s is *reachable* if there exists an initial state s_0 such that $s_0 \to^*_\mathcal{P} s$.

Brockschmidt et al. [2] use *conditional invariants*, which like standard invariants are inductive, but not necessarily initiated in all program runs, are used for proving *conditional safety* properties. Therefore, conditional invariants fulfill the **consecution** condition (which ensures that the invariant holds after every transition in the loop), but may not fulfill the **initiation** condition (which ensures that the invariant holds for the entry transitions).

We say that a map $\mathcal{Q} : \mathcal{L} \to F(\mathcal{V})$ is a *conditional (inductive) invariant* for a program (component) $\mathcal{P}$ if for all $(\ell, \boldsymbol{v}) \to_\mathcal{P} (\ell', \boldsymbol{v'})$, we have $\boldsymbol{v} \models \mathcal{Q}(\ell)$ implies $\boldsymbol{v'} \models \mathcal{Q}(\ell')$. Given a component $\mathcal{C}$, it can be shown by induction that a map $\mathcal{Q}$ is a conditional inductive invariant for $\mathcal{C}$ if and only if $\mathcal{Q}(\ell) \wedge \tau \Rightarrow \mathcal{Q}'(\ell')$ for all $(\ell, \tau, \ell') \in \mathcal{C}$.

A program $\mathcal{P}$ is said to be *terminating* if any evaluation starting at an initial state is finite. An important tool for proving termination are *ranking functions*:

▶ **Definition 1** (Ranking Function)**.** Let $\mathcal{C}$ be a component, and $t = (\ell, \tau, \ell') \in \mathcal{C}$. A function $R : \mathcal{V} \to \mathbb{Z}$ is said to be a *ranking function* for t if: (i) $\tau \models R \geq 0$ ([**Boundedness**]); (ii) $\tau \models R > R'$ ([**Decrease**]); (iii) $\hat{\tau} \models R \geq R'$ for every $(\hat{\ell}, \hat{\tau}, \hat{\ell}') \in \mathcal{C}$ ([**Non-increase**]).

The key property of ranking functions is that if a transition admits a ranking function, then it cannot be infinitely executed.

3 Conditional Termination

The main concept in this paper is the following:

▶ **Definition 2** (Conditional Termination)**.** The program $\mathcal{P}$ is (t, φ)-*conditionally terminating* if every evaluation that contains an evaluation step $(\ell, \boldsymbol{v}) \to_t (\ell', \boldsymbol{v'})$ with $\boldsymbol{v'} \models \varphi$ only uses transitions from $\mathcal{P}$ a finite number of times. In that case we say that the assertion (t, φ) is a *precondition* for termination.

```
int main() {
  int x, y, z;
  x = nondet();
  y = nondet();
  z = nondet();
  while (y ≥ 0 && z ≠ 0) {
    if (z < 0) { y = y + z;
                 z = z - 1;
    } else { x = x - z;
             y = y + x;
             z = z + 1;
} } }
```

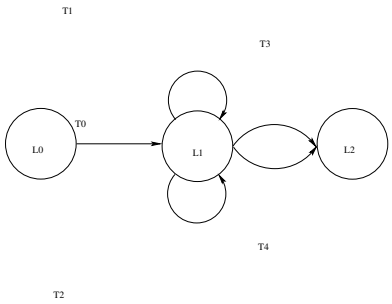


Figure 1 Program and its transition system.

Our method for generating preconditions for termination is a variation of the constraint-based approach for proving (unconditional) termination by Larraz et al. [4]. Each component $\mathcal{C}$ of the program is handled at a time, and a lexicographic conditional termination argument for it is constructed iteratively as follows. Essentially, in each iteration we synthesize a linear ranking function and possibly supporting linear conditional invariants, which show that a transition of $\mathcal{C}$ is *finitely executable*, i.e., cannot be infinitely executed. If after some iterations it has been finally proved that no transition of $\mathcal{C}$ can be infinitely executed, then the conjunction of all conditional invariants obtained at an entry location yields a precondition for termination. Indeed, once the conditional invariants hold at that entry location, then by inductiveness they hold from then on at all locations of $\mathcal{C}$, and hence the termination argument applies.

▶ **Example 3.** Consider the program in Figure 1 and its transition system, with initial location ℓ_0 and return location ℓ_2. Let us produce a precondition of termination for the component $\mathcal{C} = \{\tau_1, \tau_2\}$, corresponding to the `while` loop.

For this program, our approach could generate, e.g., the conditional invariant $z < 0$ at location ℓ_1 and the ranking function y for τ_1. Note that indeed $z < 0$ is a conditional invariant: it is preserved by τ_1 as z decreases its value, and is also trivially preserved by τ_2 since this transition is in fact disabled if $z < 0$. Moreover, y is a ranking function for τ_1, as y is bounded and decreases in τ_1 (and τ_2 is disabled).

Finally, let $t_0 = (\ell_0, \tau_0, \ell_1) \in \mathcal{E}_\mathcal{C}$ be the only entry transition of $\mathcal{C}$. Altogether, since there are no transitions left to be proved finitely executable, we get that $(t_0, z \leq 1)$ is a precondition for termination: any evaluation that contains an evaluation step $\rightarrow_{t_0} (\ell_1, \boldsymbol{v})$ with $\boldsymbol{v} \models z \leq 1$ must leave $\mathcal{C}$, and hence is finite.

Our procedure for generating preconditions for termination proceeds in a similar way as the one by Brockschmidt et al. [2] for generating precontidions for safety proofs. It takes as inputs the component $\mathcal{C}$ under consideration and its entry transitions $\mathcal{E}_\mathcal{C}$, and returns a conditional invariant $\mathcal{Q}$ that ensures that no infinite evaluation can remain within $\mathcal{C}$. In this case we keep a set $\mathcal{M} \subseteq \mathcal{C}$ of possibly infinitely executable transitions (i.e., those for which we have not proved conditional termination yet), called the *termination transition system*. As in [4], we also need to keep another transition system $\mathcal{I}$, called the *conditional invariant transition system*, which is like the original component $\mathcal{C}$, except for the addition of the conditional invariants found in previous iterations. Initially, both the termination and the conditional invariant transition systems are identical copies of the component $\mathcal{C}$.

While there are still potentially infinitely executable transitions, at each iteration we build a Max-SMT problem $\mathbb{F}$ to generate a ranking function and its supporting conditional invariants. We define templates I_ℓ for all locations ℓ in $\mathcal{C}$, corresponding to fixed-length conjunctions of linear inequalities on the program variables; i.e., I_ℓ is of the form $\bigwedge_{1\leq i\leq k}(a_i+\sum_{v\in\mathcal{V}} a_{i,v}v \leq 0)$ for some k and where the a_* are template variables that do not appear in $\mathcal{V}$. We also define a template R for a linear ranking function, i.e., R is of the form $a+\sum_{v\in\mathcal{V}} a_v v$. The formula $\mathbb{F}$ is defined as follows: $\bigwedge_{t\in\mathcal{E}_\mathcal{C}} \mathbb{I}_t \wedge \bigwedge_{t\in\mathcal{I}} \mathbb{C}_t \wedge \bigwedge_{t\in\mathcal{M}} \mathbb{N}_t \wedge \bigvee_{t\in\mathcal{M}}(\mathbb{B}_t \wedge \mathbb{S}_t)$
where $\mathbb{I}_t$, $\mathbb{C}_t$, $\mathbb{B}_t$, $\mathbb{S}_t$ and $\mathbb{N}_t$ represent respectively the **initiation**, **consecution**, **boundedness**, **decrease** and **non-increase** conditions for the transition t.

If they were all hard (i.e., must be fulfilled), the constraints $\bigwedge_{t\in\mathcal{E}_\mathcal{C}} \mathbb{I}_t \wedge \bigwedge_{t\in\mathcal{I}} \mathbb{C}_t$ would force that an invariant was obtained. In our context we need at least a conditional invariant, and prefer invariants over other conditional invariants as they will likely lead to weaker preconditions for termination. For this reason, while the constraints $\bigwedge_{t\in\mathcal{I}} \mathbb{C}_t$ are hard, in order to ensure that the solution will yield a conditional invariant, the constraints $\bigwedge_{t\in\mathcal{E}_\mathcal{C}} \mathbb{I}_t$, unlike in [4], are soft (may not be fulfilled with some cost). Thus, the solutions σ to $\mathbb{F}$ yield a linear function $\sigma(R)$ (the instantiation of the template ranking function R determined by σ) together with conditional invariants $\sigma(I_\ell)$. If no solution can be found, then the procedure gives up. Otherwise, given a solution σ to $\mathbb{F}$, since the $\sigma(I_\ell)$ are conditional invariants, they can be used to strengthen transitions $t=(\ell,\tau,\ell')$ by conjoining $\sigma(I_\ell)\wedge(\sigma(I_{\ell'}))'$ to τ, both in the conditional invariant transition system and in the termination transition system. Most importantly, we identify the subset of the transitions t from $\mathcal{M}$ for which the constraint $\mathbb{B}_t \wedge \mathbb{S}_t$ holds, and hence, $\sigma(R)$ is a ranking function, and they can be removed from $\mathcal{M}$.

In essence, this process corresponds to the step-wise construction of a lexicographic termination argument. For a location ℓ at which the component $\mathcal{C}$ is entered, the conjunction of all obtained $\sigma(I_\ell)$ is then a precondition for termination.

4 Proving Termination

A key advantage of the proposed approach for generating preconditions for termination is that it allows one to perform case analysis and focus on those parts of the program for which termination has not been guaranteed yet.

▶ **Example 4.** Let us consider again the program from Figure 1. In Example 3 it was shown that, if an evaluation ever satisfies property $z \leq 1$ at location ℓ_1, then it must be finite. Therefore, in order to prove *unconditional* termination we can assume the complement $z \geq 0$ at ℓ_1 and *narrow* the set of potentially infinite evaluations. If the transition system resulting from propagating $z \geq 0$ forward and backward terminates, we can conclude that the original system terminates too, as we will have covered all possible cases.

Figure 2 shows the transition system obtained after this narrowing step, in which transition τ_1 has been disabled and the other transitions are now stronger. Now, we can prove this system conditionally terminating again by producing the conditional invariant $x < 0$ (note that $z > 0$) and the ranking function y for τ_2. Finally, we can narrow again the transition system adding $x' \geq 0$ to τ_0 and $x \geq 0$ to τ_2 and prove it terminating with ranking function x without the need of any conditional invariant and, hence, without precondition.

In general, if a conditional termination proof has been obtained with the conditional invariant $\mathcal{Q} \neq \mathsf{None}$, by the inductiveness of $\mathcal{Q}$, an evaluation that satisfies $\mathcal{Q}(\ell)$ for a certain ℓ in $\mathcal{C}$ cannot remain within $\mathcal{C}$ infinitely. Hence we only need to consider evaluations such that whenever a location ℓ in $\mathcal{C}$ is reached, we have that $\mathcal{Q}(\ell)$ does not hold. Thus, the

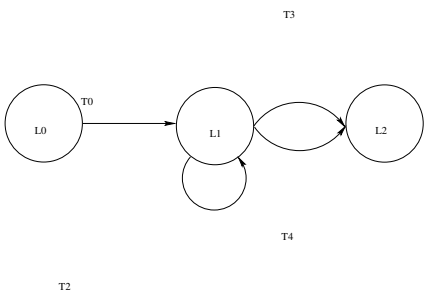


Figure 2 Transition system after narrowing.

```
int main() {
  int x = nondet();
  int y = nondet();
  assume(x > y && y ≥ 0);
  while (y > 0) {
    x = x - 1;
    y = y - 1;
  }
  while (y < 0)
    y = y + x;
}
```

Figure 3 Program that cannot be proved terminating with the approach in [4].

relation τ of an entry transition $t = (\ell, \tau, \ell') \in \mathcal{E}_{\mathcal{C}}$ can be replaced by $\tau \wedge \neg\, \mathcal{Q}(\ell')'$. And similarly, if $t = (\ell, \tau, \ell') \in \mathcal{C}$ then we can replace τ by $\tau \wedge \neg\, \mathcal{Q}(\ell) \wedge \neg\, \mathcal{Q}(\ell')'$.

▶ **Example 5.** Let us consider the program in Figure 3. This example cannot be expected to be proved terminating with the techniques presented by Larraz et al. [4] because in that work components are analyzed following a topological ordering, and no backtracking is allowed: by the time the second loop is analyzed for termination, the first one has already been proved terminating, most likely without having generated the invariant $x \geq 1$, which is needed for the second loop. On the other hand, the approach proposed here is able to successfully handle this program. Indeed, the first loop could be proved terminating with y as a ranking function. As regards the second loop, the conditional invariant $x \geq 1$ together with the ranking function $-y$ could prove it terminating. Finally, a safety checker ensures that condition $x \geq 1$ holds between the two loops.

5 Experimental Results

Termination through conditional termination has been successfully implemented in the `VeryMax` tool (`www.cs.upc.edu/~albert/VeryMax.html`). We have performed a first experimental evaluation on the set of benchmarks in the C Integer programs category of the Termination Competition which contains 335 programs. In the 2015 competition `AProVE` proved termination of 208 programs, `HipTNT+` 210 and `UltimateBuchiAutomizer` 207, while the first implementation within `VeryMax` can prove 213. Note that, we cannot handle 5 programs that include non linear expressions which can be proved by the first two other tools.

References

1 M. Bozga, R. Iosif, and F. Konecný. Deciding conditional termination. In *TACAS*, 2012. LNCS 7214, pp. 252-266. Springer.
2 M. Brockschmidt, D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Compositional Safety Verification with Max-SMT. In *FMCAD'15*, pp. 33-40, 2015.
3 B. Cook, S. Gulwani, T. Lev-Ami, A. Rybalchenko, and M. Sagiv. Proving conditional termination. In *CAV*, 2008. LNCS 5123, pp. 328-340. Springer.
4 D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Proving termination of imperative programs using Max-SMT. In *FMCAD'13*, pp. 218-225, 2013.

Certifying Safety and Termination Proofs for Integer Transition Systems*

Marc Brockschmidt[1], Sebastiaan J.C. Joosten[2], René Thiemann[2], and Akihisa Yamada[2]

1 **Microsoft Research Cambridge, UK**
2 **University of Innsbruck, Austria**

1 Introduction

A number of recently introduced techniques for proving termination of programs in languages such as Java [7, 8] and C [2, 4, 5] rely on a two-step process, in which the input program is first transformed into an intermediate formal language, and then a standard termination analyzer is used on the intermediate program. These intermediate languages are usually variations of integer transition systems (ITSs), reflecting the pervasive use of built-in integer data types in programming languages. For example, the C program in Figure 1 is translated to the ITS in Figure 2.

```
while (y >= -3)
    y = y - 2;
while (x >= 0) {
    x = x + y;
    y = y - 1;
    }
```

Figure 1 Input C program

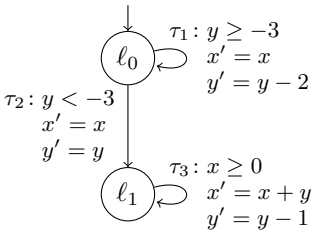


Figure 2 ITS $\mathcal{P}$, the input program as an ITS

Thus, to establish *trustworthiness* of such proofs of termination, two problems need to be tackled. First, the soundness of the translation from the source programming language needs to be proven, using elaborate models capturing the semantics of advanced programming languages. Then, the soundness of termination proofs on ITSs needs to be proven.

In this work, we tackle the second problem by discussing ongoing work to automatically *certify* termination proofs generated for a given ITS. While this continues work on certification of termination proofs of term rewrite systems [9], proving termination of ITSs requires substantially different techniques and introduces new challenges. Most notably, these include the handling of integers, the existence of designated start states of the computation, and the need to support program invariants. To generate program invariants needed for termination proofs, a number of approaches reduce the termination analysis problem to a sequence of program safety problems [1, 3, 10], which are passed to an underlying safety prover.

Thus, we first discuss how to certify safety proofs for a generalization of ITSs in Sect. 3, and then present our ongoing work on certifying termination proofs on top of this in Sect. 4.

* This work was partially supported by FWF project Y757. The authors are listed in alphabetical order regardless of individual contributions or seniority.

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 4; pp. 4:1–4:5.

2 Logic Transition Systems

Instead of ITSs, we consider a more general subset of labeled transition systems in which the actions are defined by formulas whose syntax and semantics are specified as follows:

▶ **Definition 1.** A *logic* Λ specifies a typed signature Σ and its interpretation. We denote by $\Lambda_\sigma(V)$ the set of terms of type σ over typed variables from V. An *assignment* α on V assigns each variable in V a value from the corresponding domain.

Further, we assume a type bool and $\wedge : \mathsf{bool} \times \mathsf{bool} \to \mathsf{bool} \in \Sigma$, interpreted as usual. We call a term $\phi \in \Lambda_{\mathsf{bool}}(V)$ a *formula*, and write $\alpha \models \phi$ if ϕ is interpreted to true under assignment α, and $\phi \models \psi$ to denote semantic entailment.

▶ **Definition 2.** A *logic transition system (LTS)* is a tuple $(\Lambda, \mathcal{V}, \mathcal{L}, \ell_0, \mathcal{P})$, where Λ is a logic, $\mathcal{V}$ is the set of *program variables*, $\mathcal{L}$ is the set of *locations*, $\ell_0 \in \mathcal{L}$ is the *initial location*, and $\mathcal{P}$ is the set of transition rules. Here, a *transition rule* is a triple of $\ell, r \in \mathcal{L}$ and a formula $\phi \in \Lambda_{\mathsf{bool}}(\mathcal{V} \cup \mathcal{X} \cup \mathcal{V}')$, written $\ell \xrightarrow{\phi} r$, where $\mathcal{X}$ is a set of auxiliary variables, $\mathcal{V}'$ is the set $\{v' \mid v \in \mathcal{V}\}$, and v' is a new variable called *post* variable for v.

Concerning notation, we write t' (resp. ϕ') for term t (resp. formula ϕ) where all variables v are replaced by v', and we often just write $\mathcal{P}$ for the whole LTS. Note that an LTS can be seen as a labeled transition system, which is usually abbreviated also to LTS.

A *state* is a pair (ℓ, α) of a location $\ell \in \mathcal{L}$ and an assignment α on $\mathcal{V}$. Every assignment α gives rise to an *initial state* (ℓ_0, α). There is a *transition step* from $s = (\ell, \alpha)$ to $t = (r, \beta)$, written $s \to_\mathcal{P} t$, iff $\ell \xrightarrow{\phi} r \in \mathcal{P}$ and $\alpha \cup \beta' \cup \gamma \models \phi$, where β' is the assignment on $\mathcal{V}'$ defined by $\beta'(v') = \beta(v)$, and γ is an arbitrary assignment on the auxiliary variables. If there is a transition sequence $(\ell_0, \alpha_0) \to_\mathcal{P} \cdots \to_\mathcal{P} (\ell_n, \alpha_n)$, then the state (ℓ_n, α_n) and the location ℓ_n is said to be *reachable*.

We define ITSs by fixing Λ to the integer arithmetic, i.e., there is a type int whose domain is $\mathbb{Z}$, and constants, addition, multiplication, (in)equalities etc. are in the signature.

3 Certifying Safety Proofs

The safety of a program means that certain "bad" states cannot be reached, and is usually modeled by adding a single error location $\ell_\lightning \in \mathcal{L}$ that is reached from such bad states. Proving safety then reduces to showing that $\ell_\lightning$ is not reachable.

Safety provers typically work by finding inductive invariants that show the error location is unreachable, or equivalently, that the invariant 'false' holds for the error location. To find such invariants, safety provers usually transform the transition system. Hence, a certifier has to check the soundness of such transformations, i.e., that every execution in the original system can be simulated in the transformed system, besides the validity of invariants.

Our certifier supports safety proofs as produced by the Impact [6] algorithm. Given a LTS $\mathcal{P}$ with error location $\ell_\lightning$, a safety proof takes the form of a graph $\mathcal{G}$ in which nodes (ℓ, ϕ) represent all states (ℓ, α) where $\alpha \models \phi$.

▶ **Definition 3.** A graph over nodes from $\mathcal{L} \times \Lambda_{\mathsf{bool}}(\mathcal{V})$ is a valid *unwinding* for a LTS $\mathcal{P}$ if

- it contains a node (ℓ_0, true);
- every node is categorized as either a *transition node* or a *covered node*;
- for every transition node (ℓ, ϕ) and transition rule $\ell \xrightarrow{\psi} r \in \mathcal{P}$ there is an edge $(\ell, \phi) \longrightarrow (r, \chi)$, called a *transition edge*, such that $\phi \wedge \psi \models \chi'$;

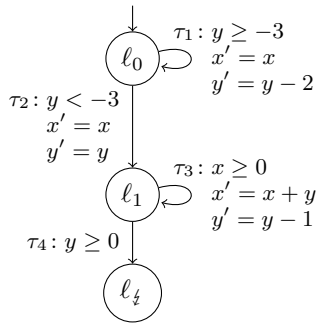


Figure 3 LTS $\mathcal{Q}$, safety ensures $y < 0$ at ℓ_1

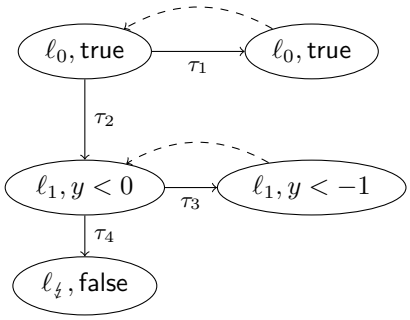


Figure 4 Graph $\mathcal{G}$, a valid unwinding of $\mathcal{Q}$

- for every covered node (ℓ, ϕ) there is an edge $(\ell, \phi) \dashrightarrow (\ell, \chi)$, called a *cover edge*, such that (ℓ, χ) is a transition node and $\phi \models \chi$;
- In every node $(\ell_\lightning, \phi)$ the formula ϕ is unsatisfiable, i.e., $\phi \models \mathsf{false}$.

▶ **Example 4.** Consider again the LTS $\mathcal{P}$ of Figure 2. In order to prove the termination of the second while loop, the invariant $y < 0$ in location ℓ_1 is essential. To verify this invariant, we create a copy of $\mathcal{P}$ as $\mathcal{Q}$, which additionally contains a transition $\ell_1 \xrightarrow{y \geq 0} \ell_\lightning$, cf. Figure 3. Clearly, if $\mathcal{Q}$ is safe, then $y < 0$ holds at location ℓ_1.

To ensure the safety of $\mathcal{Q}$, graph $\mathcal{G}$ in Figure 4 is constructed using the Impact algorithm [6]. Here, the node $\ell_1, y < -1$ is a "covered node", whose safety is proven by referring to node $\ell_1, y < 0$, which "covers" all described program states. Checking validity demands several entailment checks. For instance, for the edge for transition τ_3 we need to ensure

$$\underbrace{y < 0}_{\phi} \wedge \underbrace{x \geq 0 \wedge x' = x + y \wedge y' = y - 1}_{\psi} \models \underbrace{y' < -1}_{\chi'}.$$

We have formally proven that the existence of a valid unwinding ensures safety in Isabelle/HOL.

▶ **Theorem 5** (In Isabelle/HOL). *If $\mathcal{G}$ is a valid unwinding for $\mathcal{P}$, then $\mathcal{P}$ is safe.*

We model unwindings basically following the original definition [6], where nodes and transition edges are specified as a tree, whereas cover edges are given as a separate set. However, this turned out to be unwieldy in the formalization. Our formalization is not restricted to trees (since being a tree or not is irrelevant for soundness), each node has either exactly one cover edge or a list of transition edges, nodes are modeled by some parametric type α, and the location and the invariant of a node are modeled by a function from α to $\mathcal{L} \times \Lambda_{\mathsf{bool}}(\mathcal{V})$.

Whereas Theorem 5 is a statement about the soundness of the technique in [6], we also implemented an executable certifier for safety proofs. It demands that $\mathcal{G}$ is provided in the certificate and validates the various entailments $\phi \models \chi$ within Definition 3. To this end, the certificate also has to contain hints on how to prove each of the entailments.

Most of the formalization of the certifier is generic, i.e., it is not restricted to *integer* transition systems. However, at the moment we only have a formalized entailment checker for linear integer arithmetic. Hence, we arrive at the following soundness theorem for CeTA's safety certifier.

▶ **Theorem 6** (In Isabelle/HOL). *Let $\mathcal{P}$ be an LTS over linear integer formulas. If the certifier accepts a certificate for $\mathcal{P}$, then $\mathcal{P}$ is safe.*

Under `http://cl-informatik.uibk.ac.at/~thiemann/ceta_safety.tgz` we provide a version of CeTA for validating safety proofs. The archive also contains a small hand-written safety proof following Figures 1 and 2 of [6], as well as an automatically generated safety proof by T2 – similar to Figure 4. To this end, we employed the T2 version that is available at `https://github.com/mmjb/T2/tree/cert`.

4 Towards Certifying Termination Proofs

▶ **Definition 7.** An LTS $\mathcal{P}$ is *terminating* if there exist no infinite transition sequence starting from the initial location: $(\ell_0, \alpha_0) \to_{\mathcal{P}} (\ell_1, \alpha_1) \to_{\mathcal{P}} \cdots$.

The cooperation graph technique [1] reduces termination proving to safety checking, and incorporates some insights from termination proving for TRSs, such as deletion of rules in the dependency pair setting.

For certification purpose, it turns out that the full cooperation graph technique need not be formalized; instead, the following notion suffices.

▶ **Definition 8** (Cooperation Problem)**.** We define a set of fresh locations $\mathcal{L}^\sharp = \{\ell^\sharp \mid \ell \in \mathcal{L}\}$. A *cooperation problem* $\mathcal{Q}$ is an LTS over $\mathcal{L} \cup \mathcal{L}^\sharp$ such that every transition rule in $\mathcal{Q}$ is either of form $\ell \xrightarrow{\phi} r$, $\ell \xrightarrow{\phi} r^\sharp$, or $\ell^\sharp \xrightarrow{\phi} r^\sharp$ for $\ell, r \in \mathcal{L}$. We say the cooperation problem is *terminating* if there exists no infinite sequence of form

$$(\ell_0, \alpha_0) \to_{\mathcal{Q}} \cdots \to_{\mathcal{Q}} (\ell_n, \alpha_n) \to_{\mathcal{Q}} (\ell_n^\sharp, \alpha_n) \to_{\mathcal{Q}} (\ell_{n+1}^\sharp, \alpha_{n+1}) \to_{\mathcal{Q}} \cdots$$

where each transition rule $\ell^\sharp \xrightarrow{\phi} r^\sharp$ used after the n-th step must be used infinitely often.

▶ **Theorem 9** (In Isabelle/HOL)**.** *Let $\mathcal{P}$ be an LTS and $\mathcal{Q}$ a cooperation problem such that*

- *for each location $\ell \in \mathcal{L}$, there is a transition rule $\ell \xrightarrow{\phi} \ell^\sharp \in \mathcal{Q}$ where ϕ is a conjunction of identities $x' = x$; and*
- *for each transition rule $\ell \xrightarrow{\phi} r \in \mathcal{P}$, there exist $\ell \xrightarrow{\phi} r \in \mathcal{Q}$ and $\ell^\sharp \xrightarrow{\phi} r^\sharp \in \mathcal{Q}$.*

Then, if $\mathcal{Q}$ is terminating w.r.t. Definition 8, then $\mathcal{P}$ is terminating w.r.t. Definition 7.

The crucial advantage of considering cooperation problems is that one can remove a transition rule $\ell^\sharp \xrightarrow{\phi} r^\sharp$ without affecting termination if the rule cannot be applied infinitely often. This is unsound for original LTSs; consider e.g. a nonterminating LTS consisting of only the two transition rules $\ell_0 \xrightarrow{\mathsf{true}} \ell_1$ and $\ell_1 \xrightarrow{\mathsf{true}} \ell_1$. Clearly, the first transition rule can be applied only once. Nevertheless, if one removes it then ℓ_1 becomes unreachable, and the resulting LTS is terminating.

We are now able to formalize the main termination procedure for cooperation problems, namely transition rule removal with invariants and rank functions. As a first step we only formalize it with $\mathbb{Z}$ as target domain.

▶ **Theorem 10** (In Isabelle/HOL)**.** *Let $\mathcal{P}$ be a cooperation problem over $\mathcal{L} \cup \mathcal{L}^\sharp$. Let $I : \mathcal{L}^\sharp \to \Lambda_{\mathsf{bool}}(\mathcal{V})$ map locations to* invariants*, $R : \mathcal{L}^\sharp \to \Lambda_{\mathsf{int}}(\mathcal{V})$ map locations to* rank functions *(encoded as (linear) integer expressions), $\mathcal{D}$ a set of transition rules and $b \in \mathbb{Z}$ such that*

- *The invariants specified by I are valid, i.e., $\beta \models I(\ell^\sharp)$ whenever $(\ell_0, \alpha) \to_{\mathcal{P}}^* (\ell^\sharp, \beta)$.*
- *For every $\ell^\sharp \xrightarrow{\phi} r^\sharp \in \mathcal{P}$, $I(\ell^\sharp) \wedge \phi \models R(\ell^\sharp) \geq R(r^\sharp)'$.*
- *Transition rules in $\mathcal{D}$ are of the form $\ell^\sharp \xrightarrow{\phi} r^\sharp$ and $I(\ell^\sharp) \wedge \phi \models R(\ell^\sharp) > R(r^\sharp)' \wedge R(\ell^\sharp) \geq b$.*

Then the termination of $\mathcal{P}\setminus\mathcal{D}$ implies the termination of $\mathcal{P}$ (w.r.t. Definition 8), where $\mathcal{P}\setminus\mathcal{D}$ denotes the cooperation problem obtained by removing of all transitions rules in $\mathcal{D}$ from $\mathcal{P}$.

▶ **Example 11.** Consider again the LTS $\mathcal{P}$ of Example 4. We first construct the initial cooperation problem which yields copied versions $\tau_1^\sharp, \tau_2^\sharp, \tau_3^\sharp$ of transition rules τ_1, τ_2, τ_3.

One usually would delete transition $\tau_2^\sharp$ via an SCC-analysis, but this can also be mimicked by Theorem 10: choose $R(\ell_0^\sharp) = 1$, $R(\ell_1^\sharp) = 0$, $b = 0$, and $I(\ell_i^\sharp) = \mathsf{true}$.

Transition $\tau_1^\sharp$ corresponding to the first while loop can also be deleted without invariants: choose $R(\ell_0^\sharp) = y$, $R(\ell_1^\sharp) = 0$, $b = -3$, and $I(\ell_i^\sharp) = \mathsf{true}$.

Finally, transition $\tau_3^\sharp$ demands the invariant from Example 4. We choose $R(\ell_1^\sharp) = x$, $b = 0$, and $I(\ell_1^\sharp) = y < 0$. Hence, besides the invariant one has to validate the entailment

$$\underbrace{y < 0}_{I(\ell_1^\sharp)} \wedge \underbrace{x \geq 0 \wedge x' = x + y \wedge y' = y - 1}_{\phi} \models \underbrace{x}_{R(\ell_1^\sharp)} > \underbrace{x'}_{R(\ell_1^\sharp)'} \wedge \underbrace{x}_{R(\ell_1^\sharp)} \geq \underbrace{0}_{b}.$$

5 Conclusion

Certification establishes trustworthiness of termination and safety proofs of integer transition systems. In our formally verified certifier CeTA, we implemented a mode that certifies safety proofs for ITSs.

We moreover made a fundamental step towards certifying termination proofs for ITSs. Future work includes deciding a machine-readable format of ITS termination proofs, building a parser, and establishing a connection to the safety proof certifier. The last is required in order to certify termination proofs that use invariants. A valid unwinding $\mathcal{G}$ for an LTS $\mathcal{P}$ is used to get the invariants mentioned in Theorem 10, which is the main connection that is still missing for the termination certifier.

References

1 Marc Brockschmidt, Byron Cook, and Carsten Fuhs. Better termination proving through cooperation. In *CAV'13*.

2 Marc Brockschmidt, Byron Cook, Samin Ishtiaq, Heidy Khlaaf, and Nir Piterman. T2: Temporal property verification. In *TACAS'16*.

3 Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *PLDI'06*.

4 Stephan Falke, Deepak Kapur, and Carsten Sinz. Termination analysis of C programs using compiler intermediate languages. In *RTA'11*.

5 Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Proving termination of imperative programs using max-SMT. In *FMCAD'13*.

6 Ken McMillan. Lazy abstraction with interpolants. In *CAV'06*.

7 Carsten Otto, Marc Brockschmidt, Christian von Essen, and Jürgen Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *RTA'10*.

8 Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyser for Java Bytecode based on path-length. *ACM Transactions on Programming Languages and Systems*, 32(3), 2010.

9 René Thiemann and Christian Sternagel. Certification of termination proofs using CeTA. In *TPHOLs'09*.

10 Caterina Urban, Arie Gurfinkel, and Temesghen Kahsai. Synthesizing ranking functions from bits and pieces. In *TACAS'16*.

Automated Inference of Upper Complexity Bounds for Java Programs

F. Frohn[1], M. Brockschmidt[2], and J. Giesl[1]

1 **LuFG Informatik 2, RWTH Aachen University, Germany**
2 **Microsoft Research, Cambridge, UK**

Abstract

We report on our ongoing work on automated runtime complexity analysis of Java programs. Our technique transforms Java programs to integer transition systems and analyzes the complexity of the resulting systems using existing tools. To obtain a precise transformation, we construct a *symbolic execution graph* from the analyzed Java program and exploit the information represented in this graph. We implemented the presented technique in our tool AProVE and evaluated its power on the Termination Problem Data Base.

1998 ACM Subject Classification D.2.4 - Software/Program Verification, F.1.3 - Complexity Measures and Classes, F.3.1 - Specifying and Verifying and Reasoning about Programs

Keywords and phrases Runtime Complexity Analysis, Java, Program Verification

1 Introduction

In this paper, we report on ongoing work to extend our techniques from [3, 4, 7] (for termination analysis of Java) in order to analyze the complexity of (recursion-free) Java programs. As in our approach for termination, our goal is to transform Java programs to rewrite systems and then re-use existing tools to analyze the resulting systems. However, in contrast to termination analysis, we do not transform Java programs to term rewrite systems (with built-in integers), but to integer transition systems. One reason for this decision is that existing techniques for termination analysis of term rewrite systems with built-in integers [5] were not yet lifted to complexity analysis.

The challenges of this transformation are related to the handling of recursively defined data structures like lists or trees. To deal with such data structures, we need a measure of *size* for Java objects. While the *path length abstraction* [1, 8] results in unbounded runtime for many realistic Java programs, the *term size abstraction* (see, e.g., [6, 9]) is not directly applicable to Java objects, as these may contain (negative) integers as values. Hence, we use an adaptation of term size, cf. Sect. 2.

Furthermore, we have to infer bounds on the effects that Java instructions accessing the heap have on our data measures. These effects are difficult to estimate in the presence of sharing or cyclic data objects, cf. Sect. 3. To solve this, we exploit the detailed information about sharing and heap shapes from the *symbolic execution graph* [4] which is obtained by a whole-program analysis which over-approximates all possible program runs.

Finally, we explain how to adapt our approach to infer other forms of bounds in Sect. 4 and conclude in Sect. 5.

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 5; pp. 5:1–5:5.

```
class List {
  int value;
  List next;

  static void g(List l) {
    int head = l.value;
    while (head > 0) head--;
  }
}
```

Figure 1 A program accessing the value of the first list element

2 Measuring the Size of Java Objects

As an example,[1] consider the Java class in Fig. 1. Intuitively, the complexity of the method `g` is linear. However, at a second glance the situation is less clear. The reason is that there are several possible *size measures* $\|\,.\,\|$ for the method's argument `l`. One common definition for a measure of lists is the following:[2]

$$\|\texttt{l}\| := \begin{cases} 0 & \text{if } \texttt{l} = \texttt{null} \\ 1 + \|\texttt{l.next}\| & \text{otherwise} \end{cases} \tag{1}$$

In this way, the size of the list `l` is measured by its length, corresponding to the well-known path length abstraction [1, 8]. However, this definition of $\|\,.\,\|$ is not well defined for cyclic objects (e.g., if $\texttt{l.next} = \texttt{l}$ holds). To overcome this problem, let $\mathsf{reach}(\texttt{l})$ be the set of all objects reachable from `l` (including `l` itself). Then the definition

$$\|\texttt{l}\| := \begin{cases} 0 & \text{if } \texttt{l} = \texttt{null} \\ |\mathsf{reach}(\texttt{l})| & \text{otherwise} \end{cases} \tag{2}$$

is equivalent to (1) for acyclic lists, but also measures the length of cyclic lists correctly. Both of these measures focus on the *structure* of the measured object, but do not consider (primitive) values stored in it. Thus, the value of `l.value` is not reflected in $\|\texttt{l}\|$, and we cannot express an upper bound on the size of `head` in terms of $\|\texttt{l}\|$. Hence, the runtime complexity of `g` would be unbounded. Therefore, we now use the *term size* of `l` by defining

$$\|\texttt{l}\| := \begin{cases} 0 & \text{if } \texttt{l} = \texttt{null} \\ |\mathsf{reach}(\texttt{l})| + \sum_{\texttt{x} \in \mathsf{reach}(\texttt{l})} \|\texttt{x.value}\| & \text{otherwise} \end{cases} \tag{3}$$

Here, one has to fix a suitable size measure for integers. Obviously, the integer's value itself is not a good choice: If, e.g., `l` is the list $[10, -100]$, then we get $\|\texttt{l}\| < 0$ and, again, cannot express an upper bound on `head` in terms of the size of `l`. Hence, as in [2], we measure integers by their absolute value, i.e., we fix $\|i\| = |i|$ for all $i \in \mathbb{Z}$.

To define our size measure for arbitrary objects, let $\mathsf{prim}(\texttt{o})$ be the set of all fields of (primitive) integer types[3] of the object `o`. So in our example, we have $\mathsf{prim}(\texttt{l}) = \{\texttt{value}\}$.

[1] Note that our implementation analyzes Java Bytecode instead of Java source code, i.e., Java programs have to be compiled in order to analyze their complexity.
[2] For the sake of simplicity, we sometimes identify program variables and the objects referenced by them.
[3] Integer types are types like `int` and `long`, but also `boolean`, as booleans are internally represented as integers by the JVM.

```
while (l != null) l = l.next;
```

Figure 2 A list traversal program

Then we define:

$$\|\mathtt{o}\| = \begin{cases} 0 & \text{if } o = \mathtt{null} \\ |\mathsf{reach}(\mathtt{o})| + \sum_{\mathtt{x} \in \mathsf{reach}(\mathtt{o}),\ \mathtt{f} \in \mathsf{prim}(\mathtt{x})} |\mathtt{x.f}| & \text{otherwise} \end{cases} \tag{4}$$

Hence, we measure objects as the number of all reachable objects plus the absolute values of all reachable integers.

3 Inferring Bounds for Heap Accesses

One important motivation for the choice of our size measure (cf. Sect. 2) was the ability to infer bounds for functions like `g` from Fig. 1. To this end, we have to be able to relate the size of `head` to the size of the list `l`. Indeed, with our size measure, the following holds for read accesses to fields `f` of integer types:

$$\text{if the field } \mathtt{f} \text{ of } \mathtt{x} \text{ is of an integer type, then we have } -\|\mathtt{x}\| < \mathtt{x.f} < \|\mathtt{x}\| \tag{5}$$

In this way, the function `g` from Fig. 1 can be translated into the following integer transition system, whose linear runtime complexity can easily be shown by existing tools:

$$\begin{array}{rcll} g(l) & \to & g_{while}(head) & [\![-l < head < l]\!] \\ g_{while}(head) & \to & g_{while}(head - 1) & [\![head > 0]\!] \end{array}$$

Similarly, the following observation for fields `f` of reference types allows us to prove upper complexity bounds for list- and tree-traversal algorithms:

$$\text{if } \mathtt{x} \text{ is acyclic, then we have } 0 \leq \|\mathtt{x.f}\| < \|\mathtt{x}\| \text{ for all fields } \mathtt{f} \text{ of } \mathtt{x} \tag{6}$$

Note that, due to the extensions of [3], the symbolic evaluation graph contains information about the cyclicity of data structures and hence can be used to check the premise of (6). As a result, the loop from Fig. 2 can be translated to the following integer transition system if `l` is an acyclic list:

$$g_{while}(l) \to g_{while}(l') \quad [\![0 \leq l' < l]\!]$$

Again, existing tools can easily prove a linear upper bound for this integer transition system.

If we fail to prove that an object is acyclic, then we can still use the following observation to obtain size bounds for its successors:

$$\text{we always have } 0 \leq \|\mathtt{x.f}\| \leq \|\mathtt{x}\| \text{ for all fields } \mathtt{f} \text{ of } \mathtt{x} \tag{7}$$

So if `l` is a cyclic list, then the loop from Fig. 2 does not terminate and it is transformed to the following non-terminating integer transition system:

$$g_{while}(l) \to g_{while}(l') \quad [\![0 \leq l' \leq l]\!]$$

As in the case of read accesses to the heap, we can also specify bounds on the size of objects after write accesses. Here, we have to take sharing effects into account. Let

$\mathsf{pred}(\mathsf{y}) = \{\mathsf{o} \mid \mathsf{y} \in \mathsf{reach}(\mathsf{o})\}$, i.e., $\mathsf{pred}(\mathsf{y})$ is the set of all predecessors of y. In this way, $\mathsf{pred}(\mathsf{y})$ contains all objects whose size is affected by write accesses to fields of y. Then the following observation allows us to over-approximate the size of each object $\mathsf{o} \in \mathsf{pred}(\mathsf{y})$:

$$\text{if } k = \|\mathsf{o}\| \text{ before evaluating } \texttt{y.f = x}, \text{then afterwards we have } 0 \leq \|\mathsf{o}\| \leq k + \|\mathsf{x}\| \quad (8)$$

Note that the set $\mathsf{pred}(\mathsf{y})$ can easily be over-approximated using the information about sharing and aliasing from the symbolic execution graph.

4 Beyond Time Complexity

By applying the abstraction described in Sect. 2 and handling accesses to the heap as explained in Sect. 3, we obtain an integer transition system which closely mirrors the analyzed Java program and which can be used to analyze its time complexity. To analyze the usage of other resources like, e.g., space complexity or network traffic instead, we proceed as in [2] and attach *weights* to the rules of the integer transition system. In particular, this allows to infer bounds when only *certain* instructions need to be considered. For example, to measure space complexity, each transition corresponding to a `new` instruction has weight 1 and all other[4] transitions have weight 0. Similarly, we can apply our technique to analyze various other properties of Java programs.

The power of the resulting analysis can be significantly improved using the following observation: If a loop ℓ of the analyzed program has weight 0 and no other loop ℓ' with positive weight and no other transition t with non-constant weight is reachable from ℓ, then ℓ can safely be removed from the analyzed program without affecting its asymptotic complexity. In this way, the resulting integer transition systems can often be simplified substantially.

5 Conclusion

We presented the basic ideas of our ongoing work towards a transformation of Java programs to integer transition systems which is suitable for complexity analysis. One of its essential properties is the used size abstraction (cf. Sect. 2), which allows us to prove complexity bounds for many programs where techniques based on the path-length abstraction fail. Moreover, due to the information about heap shapes and sharing from the symbolic execution graph we can handle heap-manipulating programs with user-defined recursive data structures. Apart from time complexity, we outlined how to adapt our transformation for the analysis of other forms of complexity.

We implemented the transformation of Java programs to integer transition systems in our tool AProVE[5] and evaluated it on the Termination Problem Data Base.[6] To analyze the resulting integer transition systems, we use the tool KoAT [2]. Currently, AProVE is able to prove polynomial upper bounds for 151 of the 300 examples from the category `Java_Bytecode` (i.e., we excluded the examples from the category `Java_Bytecode_Recursive`, as our technique for complexity analysis was not yet lifted to recursive programs). Note that at least 83 of these 300 examples do not terminate, as their non-termination can be proved by AProVE. Hence, our techniques succeeds for 70% of the remaining 217 examples. For space complexity, AProVE is even able to prove polynomial bounds for 268 of the 300 examples (89%).

[4] Here, we ignore arrays and the instructions to create them for the sake of simplicity.
[5] `http://aprove.informatik.rwth-aachen.de/`
[6] `http://termination-portal.org/wiki/TPDB`

Acknowledgments This research is supported by the DFG grant GI 274/6-1 and the Air Force Research Laboratory (AFRL).

References

1 Elvira Albert, Puri Arenas, Samir Genaim, Germán Puebla, and Damiano Zanardini. COSTA: Design and implementation of a cost and termination analyzer for Java Bytecode. In *FMCO '07*, LNCS 5382, pages 113–132, 2008.

2 Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Analyzing runtime and size complexity of integer programs. *ACM Transactions on Programming Languages and Systems*, 38(4), 2016. Preliminary version appeared in *TACAS '14*, LNCS 8413, pages 140–155, 2014.

3 Marc Brockschmidt, Richard Musiol, Carsten Otto, and Jürgen Giesl. Automated termination proofs for Java programs with cyclic data. In *CAV '12*, LNCS 7358, pages 105–122, 2012.

4 Marc Brockschmidt, Carsten Otto, Christian von Essen, and Jürgen Giesl. Termination graphs for Java Bytecode. In *Verification, Induction, Termination Analysis*, LNCS 6463, pages 17–37, 2010.

5 Carsten Fuhs, Jürgen Giesl, Martin Plücker, Peter Schneider-Kamp, and Stephan Falke. Proving termination of integer term rewriting. In *RTA '09*, LNCS 5595, pages 32–47, 2009.

6 Samir Genaim, Michael Codish, John P. Gallagher, and Vitaly Lagoon. Combining Norms to Prove Termination. In *VMCAI '02*, LNCS 2294, pages 126–138, 2002.

7 Carsten Otto, Marc Brockschmidt, Christian von Essen, and Jürgen Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *RTA '10*, LIPIcs 10, pages 259–276, 2010.

8 Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyzer for Java Bytecode based on path-length. *ACM Transactions on Programming Languages and Systems*, 32(3), 2010.

9 Florian Zuleger, Sumit Gulwani, Moritz Sinn, and Helmut Veith. Bound analysis of imperative programs with the size-change abstraction. In *SAS '11*, LNCS 6887, pages 280–297, 2011.

Lower Runtime Bounds for Integer Programs*

F. Frohn[1], M. Naaf[1], J. Hensel[1], M. Brockschmidt[2], and J. Giesl[1]

1 **LuFG Informatik 2, RWTH Aachen University, Germany**
2 **Microsoft Research, Cambridge, UK**

Abstract

We present a technique to infer *lower* bounds on the worst-case runtime complexity of integer programs. To this end, we use under-approximating program simplification techniques and deduce asymptotic lower bounds from the resulting simplified programs. We implemented our technique in our tool LoAT and show that it infers non-trivial lower bounds for a large number of examples.

1998 ACM Subject Classification D.2.4 - Software/Program Verification, F.1.3 - Complexity Measures and Classes, F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Runtime Complexity Analysis, Integer Programs, Lower Bounds

1 Introduction

Recently, efficient methods were developed to find upper bounds on the worst-case complexity of integer programs [1,2,5,7,12]. To infer tight complexity bounds, lower bounds for this notion of complexity are required as well. Such lower bounds also have important applications in security analysis. For example, large (e.g., exponential) lower bounds can be used to identify denial-of-service attacks that exploit the algorithmic worst-case complexity of a program. However, up to now there are only a few approaches to deduce *best-case lower bounds* [1,3] and no work on *worst-case lower bounds* for integer programs. For term rewrite systems, we recently introduced the first technique to infer worst-case lower bounds [8].

In this paper, we consider worst-case lower bounds for integer programs (see [9] for the full version of our paper). Let $\mathcal{A}(\mathcal{V})$ be the set of arithmetic terms over the variables $\mathcal{V}$ and let $\mathcal{F}(\mathcal{V})$ be the set of conjunctions of (in)equations over $\mathcal{A}(\mathcal{V})$. We represent integer programs as directed graphs where nodes are program *locations* $\mathcal{L}$ and edges are program *transitions* $\mathcal{T}$, where $\mathcal{L}$ contains a *canonical start location* ℓ_0. W.l.o.g., no transition leads back to ℓ_0.

▶ **Definition 1** (Programs). *Configurations* $(\ell, \boldsymbol{v})$ consist of a location $\ell \in \mathcal{L}$ and a *valuation* $\boldsymbol{v} : \mathcal{V} \to \mathbb{Z}$. Valuations are lifted to terms and formulas as usual. A *transition* $t = (\ell, \gamma, \eta, c, \ell')$ can evaluate $(\ell, \boldsymbol{v})$ to $(\ell', \boldsymbol{v}')$ if $\boldsymbol{v} \models \gamma$ for the *guard* $\gamma \in \mathcal{F}(\mathcal{V})$. The *update* $\eta : \mathcal{V} \to \mathcal{A}(\mathcal{V})$ maps any $x \in \mathcal{V}$ to a term $\eta(x)$ where $\boldsymbol{v}(\eta(x)) \in \mathbb{Z}$. It determines[1] $\boldsymbol{v}'$ by setting $\boldsymbol{v}'(x) = \boldsymbol{v}(\eta(x))$. Such an *evaluation step* has *cost* $k = \boldsymbol{v}(c)$ for $c \in \mathcal{A}(\mathcal{V})$ and is written $(\ell, \boldsymbol{v}) \to_{t,k} (\ell', \boldsymbol{v}')$. We sometimes drop the indices t, k. A *program* is a set of transitions $\mathcal{T}$.

Fig. 1 shows an example, where the pseudo-code on the left corresponds to the program on the right. We write the costs of a transition in [ ] next to its name and represent updates by imperative commands. We use x to refer to the value of the variable x before the update and x' to refer to x's value after the update. Here, we have $(\ell_3, \boldsymbol{v}) \to_{t_4} (\ell_3, \boldsymbol{v}')$ for all valuations $\boldsymbol{v}$ where $\boldsymbol{v}(u) > 0$, $\boldsymbol{v}'(u) = \boldsymbol{v}(u) - 1$, and $\boldsymbol{v}'(v) = \boldsymbol{v}(v)$ for all $v \in \{x, y, z\}$.

* Supported by the DFG grant GI 274/6-1 and the Air Force Research Laboratory (AFRL).
[1] See [9] for a generalization of our program model which also allows non-deterministic assignments.

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 6; pp. 6:1–6:5.

```
ℓ0: y = 0
ℓ1: while x > 0 do
      y = y + x
      x = x − 1
    done
    z = y
ℓ2: while z > 0 do
      u = z − 1
ℓ3:   while u > 0 do
        u = u − 1
      done
      z = z − 1
    done
```

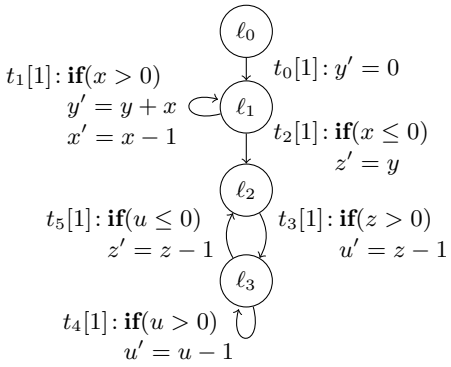


Figure 1 Example integer program

2 Simplifying Programs to Compute Lower Bounds

We now introduce our under-approximating program simplification technique. To this end, we first show how to under-estimate the number of possible iterations of a *simple loop* $t = (\ell, \gamma, \eta, c, \ell)$, i.e., how to infer a term b such that for all valuations with $\boldsymbol{v} \models \gamma$, t can be applied at least $\boldsymbol{v}(b)$ times.[2] To find such under-estimations, we use an adaptation of ranking functions [2, 11] which we call *metering functions*. For any term b, let $\eta(b)$ denote the term in which all variables are replaced according to η.

▶ **Definition 2** (Metering Function)**.** Let $t = (\ell, \gamma, \eta, c, \ell)$ be a transition. We call $b \in \mathcal{A}(\mathcal{V})$ a *metering function* for t iff the following conditions hold:

$$\neg\gamma \implies b \leq 0 \qquad (1) \qquad\qquad \gamma \implies \eta(b) \geq b - 1 \qquad (2)$$

Here, (2) ensures that $\boldsymbol{v}(b)$ decreases at most by 1 in each loop iteration, and (1) requires that $\boldsymbol{v}(b)$ is non-positive if the loop cannot be executed. Thus, the loop is executed *at least* $\boldsymbol{v}(b)$ times (i.e., b *under-estimates* t). So for the transition t_1 in the example of Fig. 1, x, $x-1$, $x-2$, ... are valid metering function. Our implementation builds upon a well-known transformation based on Farkas' Lemma [11] to find *linear* metering functions.

Loop Acceleration Given a metering function b, we can *accelerate* a simple loop. Let $t = (\ell, \gamma, \eta, c, \ell)$ and let η^n denote n applications of η. To accelerate t, we compute its *iterated* update and costs, i.e., a closed form η_{it} of η^n and an under-approximation c_{it} of $\sum_{0 \leq i < n} \eta^i(c)$. Then we replace t by $(\ell, \gamma, \eta_{\text{it}}[n/b], c_{\text{it}}[n/b], \ell)$ to summarize b iterations of t.

For $\mathcal{V} = \{x_1, \ldots, x_m\}$, the iterated update is computed by solving the recurrence equations $x^{(1)} = \eta(x)$ and $x^{(n+1)} = \eta(x)[x_1/x_1^{(n)}, \ldots, x_m/x_m^{(n)}]$ for all $x \in \mathcal{V}$ and $n \geq 1$. So for the transition t_1 from Fig. 1 we get the recurrence equations $x^{(1)} = x - 1$, $x^{(n+1)} = x^{(n)} - 1$, $y^{(1)} = y + x$, and $y^{(n+1)} = y^{(n)} + x^{(n)}$. Usually, they can easily be solved using state-of-the-art recurrence solvers [4]. (Otherwise, the loop cannot be accelerated and is simply removed from the program, which is possible since we are only interested in worst-case lower bounds.) In our example, we obtain the closed forms $\eta_{\text{it}}(x) = x^{(n)} = x - n$ and $\eta_{\text{it}}(y) = y^{(n)} = y + n \cdot x - \frac{1}{2}n^2 + \frac{1}{2}n$. We proceed similarly for the iterated cost of a transition, where we may under-approximate the solution of the recurrence equations $c^{(1)} = c$ and

[2] For simplicity, we assume $\boldsymbol{v}(b) \in \mathbb{Z}$. See [9] for a generalization which also allows, e.g., $b = \frac{x}{2}$.

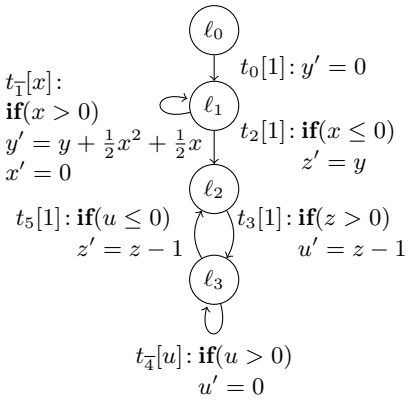


Figure 2 Accelerating t_1 and t_4

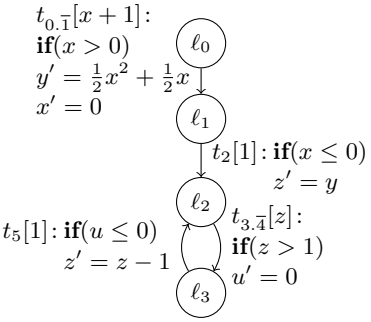


Figure 3 Eliminating $t_{\overline{1}}$ and $t_{\overline{4}}$

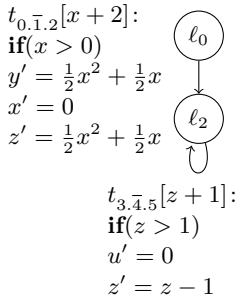


Figure 4 Eliminating ℓ_1 and ℓ_3

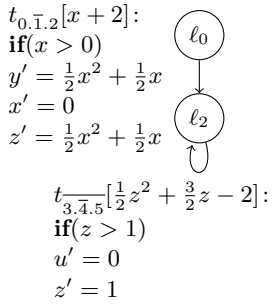


Figure 5 Accelerating $t_{3.\overline{4}.5}$

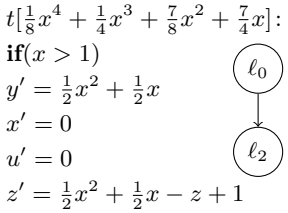


Figure 6 Eliminating $t_{\overline{3.\overline{4}.5}}$

$c^{(n+1)} = c^{(n)} + c[x_1/x_1^{(n)}, \ldots, x_m/x_m^{(n)}]$. For t_1 in Fig. 1, we get $c^{(1)} = 1$ and $c^{(n+1)} = c^{(n)} + 1$ which leads to the closed form $c_{\text{it}} = c^{(n)} = n$. In this way, in our example we obtain the program in Fig. 2 with the accelerated transitions $t_{\overline{1}}$ and $t_{\overline{4}}$.

Chaining After trying to accelerate all simple loops, we can *chain* subsequent transitions t_1, t_2 by adding a new transition $t_{1.2}$ that simulates their combination. One goal of chaining is *loop elimination* of all accelerated simple loops. To this end, we chain all subsequent transitions t', t where t is a simple loop and t' is no simple loop. Afterwards, we delete t (which is sound, as our technique is under-approximating). Moreover, once t' has been chained with all subsequent simple loops, then we also remove t', since its effect is now covered by the newly introduced (chained) transitions. So in our example from Fig. 2, we chain t_0 with $\overline{t}_1$ and t_3 with $\overline{t}_4$. The resulting program is depicted in Fig. 3.

Chaining also allows *location elimination* by chaining all pairs of incoming and outgoing transitions for a location ℓ and removing them afterwards. For Fig. 3, we chain $t_{0.\overline{1}}$ and t_2 as well as $t_{3.\overline{4}}$ and t_5 to eliminate the locations ℓ_1 and ℓ_3, leading to the program in Fig. 4.

Now $t_{3.\overline{4}.5}$ is accelerated with $b = z - 1, \eta_{\text{it}}(u) = 0$, and $\eta_{\text{it}}(z) = z - n$. To compute its iterated costs, we solve the recurrence equations $c^{(1)} = z + 1$ and $c^{(n+1)} = c^{(n)} + z^{(n)} + 1$. After computing $z^{(n)} = z - n$, the second equation simplifies to $c^{(n+1)} = c^{(n)} + z - n + 1$, which results in $c_{\text{it}} = c^{(n)} = n \cdot z - \frac{1}{2}n^2 + \frac{3}{2}n$. In this way, we obtain the program in Fig. 5. A final chaining step yields the program which is depicted in Fig. 6.

3 Asymptotic Lower Bounds for Simplified Programs

By applying a generalization of the techniques from Sect. 2 with an appropriate strategy, every

program can be transformed to a *simplified* program where all program paths have length 1. While Fig. 6 obviously witnesses the bound $\Omega(n^4)$ (where n measures the "size" of the input), in general obtaining asymptotic bounds from simplified programs is non-trivial. E.g., in Fig. 7 the costs are quadratic, but due to the condition $0 < x < 10$, $\Omega(n^2)$ is not a valid bound.

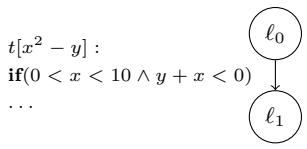


Figure 7 Simplified prog.

To infer an asymptotic bound from a transition of a simplified program, we normalize its guard γ such that it has the form $\bigwedge_{1\leq i\leq k}(a_i > 0)$. In our example we get $x > 0 \wedge 10 - x > 0 \wedge -y - x > 0$. Then we infer a valuation $\boldsymbol{v}_n$ which is parameterized in n such that $\boldsymbol{v}_n \models \gamma$ holds for large enough n. Thus, applying $\boldsymbol{v}_n$ to the transition's costs yields an asymptotic bound. Note that since the valuations $\boldsymbol{v}_n$ do not necessarily correspond to the "best" cases w.r.t. runtime, our approach only infers "worst-case" lower bounds. Obviously, $\boldsymbol{v}_n \models \gamma$ holds for large enough n if $\boldsymbol{v}_n(a_i)$ is a positive constant or increases infinitely towards ω for all a_i. Thus, we introduce a technique to find out whether fixing the valuations of some variables and increasing or decreasing the valuations of others results in positive resp. increasing valuations of $a_1, \ldots, a_k$. Our technique operates on *limit problems* $\{a_1^{\bullet_1}, \ldots, a_k^{\bullet_k}\}$ where $a_i \in \mathcal{A}(\mathcal{V})$ and $\bullet_i \in \{+, -, +_!, -_!\}$. Here, a^+ (resp. a^-) means that a grows towards ω (resp. $-\omega$) and $a^{+_!}$ (resp. $a^{-_!}$) means that a has to be a positive (resp. negative) constant. We represent the guard by an *initial limit problem* $\{a_1^{\bullet_1}, \ldots, a_k^{\bullet_k}\}$ where $\bullet_i \in \{+, +_!\}$. So $\{x^{+_!}, (10-x)^{+_!}, (-y-x)^+\}$ is an initial limit problem for the only transition of the program in Fig. 7.

A limit problem is *trivial* iff all its terms are unique variables. For trivial limit problems S we immediately obtain a family of models $\boldsymbol{v}_n^S$ by fixing

$$\begin{array}{llll} \boldsymbol{v}_n^S(x) = n, & \text{if } x^+ \in S & \boldsymbol{v}_n^S(x) = 1, & \text{if } x^{+_!} \in S \\ \boldsymbol{v}_n^S(x) = -n, & \text{if } x^- \in S & \boldsymbol{v}_n^S(x) = -1, & \text{if } x^{-_!} \in S. \end{array} \qquad \boldsymbol{v}_n^S(x) = 0, \quad \text{otherwise}$$

We now introduce a transformation $\rightsquigarrow$ to simplify limit problems until one reaches a trivial problem. If S contains $f(a_1, a_2)^\bullet$ for some standard arithmetic operation f like addition, subtraction, multiplication, division, and exponentiation, we use *limit vectors* $(\bullet_1, \bullet_2)$ with $\bullet_i \in \{+, -, +_!, -_!\}$ to characterize for which kinds of arguments f is increasing (if $\bullet = +$) resp. decreasing (if $\bullet = -$) resp. a positive or negative constant (if $\bullet = +_!$ or $\bullet = -_!$). Then S can be transformed into the new limit problem $S \setminus \{f(a_1, a_2)^\bullet\} \cup \{a_1^{\bullet_1}, a_2^{\bullet_2}\}$.

For example, $(+, +_!)$ is an increasing limit vector for subtraction. The reason is that $a_1 - a_2$ is increasing if a_1 is increasing and a_2 is a positive constant. Hence, our transformation $\rightsquigarrow$ allows us to replace $(a_1 - a_2)^+$ by a_1^+ and $a_2^{+_!}$.

Moreover, for numbers $m \in \mathbb{Z}$, one can simplify the constraints $m^{+_!}$ and $m^{-_!}$ (e.g., $2^{+_!}$ is clearly satisfied). Finally, we also allow to instantiate variables with linear arithmetic terms.

▶ **Definition 3** ($\rightsquigarrow$)**.** Let S be a limit problem. We have:

(A) $S \cup \{f(a_1, a_2)^\bullet\} \overset{\varnothing}{\rightsquigarrow} S \cup \{a_1^{\bullet_1}, a_2^{\bullet_2}\}$ if $\bullet$ is $+$ (resp. $-, +_!, -_!$) and $(\bullet_1, \bullet_2)$ is an increasing (resp. decreasing, positive, negative) limit vector for f

(B) $S \cup \{m^{+_!}\} \overset{\varnothing}{\rightsquigarrow} S$ if $m \in \mathbb{Z}$ with $m > 0$, $S \cup \{m^{-_!}\} \overset{\varnothing}{\rightsquigarrow} S$ if $m \in \mathbb{Z}$ with $m < 0$

(C) $S \overset{\sigma}{\rightsquigarrow} S\sigma$ if $\sigma : \mathcal{V} \to \mathcal{A}(\mathcal{V})$ is a substitution such that x does not occur in $x\sigma$, $x\sigma$ is linear, and $\boldsymbol{v}(x\sigma) \in \mathbb{Z}$ for all valuations $\boldsymbol{v}$ and all $x \in \mathcal{V}$

In our example, we have $\{x^{+_!}, (10-x)^{+_!}, (-y-x)^+\} \overset{\varnothing}{\rightsquigarrow} \{x^{+_!}, (10-x)^{+_!}, (-y)^+\} \overset{\varnothing}{\rightsquigarrow} \{x^{+_!}, (10-x)^{+_!}, y^-\} \overset{\{x/9\}}{\rightsquigarrow} \{9^{+_!}, 1^{+_!}, y^-\} \overset{\varnothing}{\rightsquigarrow} \{1^{+_!}, y^-\} \overset{\varnothing}{\rightsquigarrow} \{y^-\}$ using the increasing limit vectors $(+, +_!)$ and $(-)$ for subtraction and unary minus.

If we apply the substitutions $\sigma_1, \ldots, \sigma_k$ that were used during the simplification of the initial limit problem to the resulting trivial limit problem S, then $\boldsymbol{v}_n^S \circ \sigma_k \circ \cdots \circ \sigma_1$ is a

family of models for the guard of the transition under consideration. So in our example, we get $\boldsymbol{v}_n = \boldsymbol{v}_n^{\{y^-\}} \circ \{x/9\} = \{x/9, y/-n\}$. Finally, we obtain the asymptotic lower bound $\Omega(\boldsymbol{v}_n(x^2 - y)) = \Omega(81 + n) = \Omega(n)$ by applying $\boldsymbol{v}_n$ to the transition's costs.

4 Experiments

$\mathsf{rc}_{\mathcal{T}}(n)$	$\Omega(1)$	$\Omega(n)$	$\Omega(n^2)$	$\Omega(n^3)$	$\Omega(n^4)$	EXP	$\Omega(\omega)$
$\mathcal{O}(1)$	(132)	–	–	–	–	–	–
$\mathcal{O}(n)$	45	125	–	–	–	–	–
$\mathcal{O}(n^2)$	9	18	33	–	–	–	–
$\mathcal{O}(n^3)$	2	–	–	3	–	–	–
$\mathcal{O}(n^4)$	1	–	–	–	2	–	–
EXP	–	–	–	–	–	5	–
$\mathcal{O}(\omega)$	57	31	3	–	–	–	173

Our implementation LoAT ("Lower Bounds Analysis Tool") of the presented technique is freely available at [10]. LoAT uses the recurrence solver PURRS [4] and the SMT solver Z3 [6]. We evaluated LoAT on the benchmarks from the evaluation of [5]. As we know of no other tool to compute worst-case lower bounds for integer programs, we compared our results with the asymptotically smallest results of leading tools for upper bounds: KoAT [5], CoFloCo [7], Loopus [12], and RanK [2]. The results are displayed in the table on the right, where rows correspond to the best automatically inferred upper bound, and columns correspond to the worst-case lower bound computed by LoAT. A comparison with tools for best-case lower bounds would be meaningless since the worst-case lower bounds computed by LoAT are no valid best-case lower bounds. As shown in the table, LoAT infers non-trivial lower bounds for 78% of the examples. Tight bounds are proved for 67% of the examples. For a detailed experimental evaluation of our implementation and the full version [9] of our paper we refer to `http://aprove.informatik.rwth-aachen.de/eval/integerLower/`.

References

1 Albert, E., Genaim, S., Masud, A.N.: On the inference of resource usage upper and lower bounds. ACM Transactions on Computational Logic 14(3) (2013)

2 Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Proc. SAS '10. pp. 117–133. LNCS 6337 (2010)

3 Alonso-Blas, D.E., Genaim, S.: On the limits of the classical approach to cost analysis. In: Proc. SAS '12. pp. 405–421. LNCS 7460 (2012)

4 Bagnara, R., Pescetti, A., Zaccagnini, A., Zaffanella, E.: PURRS: Towards computer algebra support for fully automatic worst-case complexity analysis. CoRR abs/cs/0512056 (2005)

5 Brockschmidt, M., Emmes, F., Falke, S., Fuhs, C., Giesl, J.: Analyzing runtime and size complexity of integer programs. ACM Transactions on Programming Languages and Systems 38(4) (2016), preliminary version in Proc. TACAS '14, pp. 140–155, LNCS 8413 (2014)

6 de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proc. TACAS '08. pp. 337–340. LNCS 4963 (2008)

7 Flores-Montoya, A., Hähnle, R.: Resource analysis of complex programs with cost equations. In: Proc. APLAS '14. pp. 275–295. LNCS 8858 (2014)

8 Frohn, F., Giesl, J., Hensel, J., Aschermann, C., Ströder, T.: Inferring lower bounds for runtime complexity. In: Proc. RTA '15. pp. 334–349. LIPIcs 36 (2015)

9 Frohn, F., Naaf, M., Hensel, J., Brockschmidt, M., Giesl, J.: Lower runtime bounds for integer programs. In: Proc. IJCAR '16. pp. 550–567. LNAI 9706 (2016)

10 LoAT, available from `https://github.com/aprove-developers/LoAT`

11 Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Proc. VMCAI '04. pp. 239–251. LNCS 2937 (2004)

12 Sinn, M., Zuleger, F., Veith, H.: A simple and scalable static analysis for bound analysis and amortized complexity analysis. In: CAV '14. pp. 745–761. LNCS 8559 (2014)

SMT-Based Techniques in Automated Termination Analysis*

Carsten Fuhs[1]

1 **Birkbeck, University of London, United Kingdom**
carsten@dcs.bbk.ac.uk

Abstract

Since the early 2000s fully automated techniques and tools for termination analysis have flourished in several communities: term rewriting, imperative programs, functional programs, logic programs, ...

A common theme behind most of these tools is the use of constraint-based techniques to advance the proof of (non-)termination. Recently, in particular SAT and SMT solvers are used as back-ends to automate these techniques. In this survey, we provide an overview over automated termination analysis techniques in different communities, with an emphasis on the applied constraint-based techniques.

1998 ACM Subject Classification D.2.4 Software/Program Verification, F.3.1 Specifying and Verifying and Reasoning about Programs, F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases termination analysis, SMT solving, term rewriting, program analysis.

1 Introduction

In 1936, Turing [38] proved undecidability of the halting problem, i.e., the question whether execution of a given program would eventually come to an end. Nonetheless, in 1949 he proposed that the programmer ought to annotate the program with a proof of its termination for arbitrary inputs [39]. His suggestion was later also taken up e.g. by Floyd [11]. It involves providing a function that maps the *state* of the program during execution into a set with an associated well-founded order (e.g. $(\mathbb{N}, >)$) so that the function values decrease in $>$ during execution of the program. By well-foundedness of $>$, we can then conclude that (the reachable part of) the transition relation of the program itself must be well founded.

Fundamentally, this approach of using such *ranking functions* to prove termination has remained the core technique for proving termination. However, one aspect has changed: While in the days of Turing or Floyd a termination proof would be found by hand, in recent years it has become commonplace to search for such termination proofs automatically, via push-button tools. Nowadays the technique of choice for exploring the search space are *SAT Modulo Theories* (SMT) solvers (see, e.g., [33]). These tools can check satisfiability of (often quantifier-free) Boolean combinations of formulas over one or several first-order theories (e.g., quantifier-free linear arithmetic). The overarching theme is that a template for the termination proof step is provided by the user or a heuristic in the push-button tool, and a satisfying assignment found by the SMT solver then induces a sound termination proof (or a step in such a termination proof, showing that *certain* program executions are terminating).

* This short paper is inspired by a talk originally presented at the *Satisfiability Modulo Theories 2016* workshop in July 2016: http://smt-workshop.cs.uiowa.edu/2016/invited.shtml

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 7; pp. 7:1–7:5.

This short survey focuses on the automation of termination proofs in two communities where the development of automatic termination analysis tools has flourished: Term rewriting (Section 2) and imperative programs (Section 3).

2 Term Rewriting

Term rewriting (see, e.g., [3] for an introduction) is essentially a way of conducting equational reasoning on term algebras on syntactic level. From a programming languages perspective, term rewriting is an untyped first-order functional programming language without predefined data structures (such as integers and their usual arithmetic operations) and with non-determinism regarding both the question which subterm should be evaluated (rewritten) – the evaluation strategy – and the question which rule should be used for the rewrite step.

There exist many variations of term rewriting which drop some of the above properties. Examples are higher-order rewriting, rewriting using a specific evaluation strategy (e.g., innermost or context-sensitive [29] rewriting), or term rewriting with built-in data structures, e.g., for integer arithmetic (see, e.g., [15, 24]). However, here we consider only the basic case.

▶ **Example 1.** The following rewrite system $\mathcal{R}$ computes a function append that concatenates two lists. Here lists are built using *constructor symbols* nil for the empty list and cons for inserting an element at the head of a list.

$$\mathsf{append}(\mathsf{nil}, ys) \to ys \quad (1) \qquad \mathsf{append}(\mathsf{cons}(x, xs), ys) \to \mathsf{cons}(x, \mathsf{append}(xs, ys)) \quad (2)$$

For the term $t = \mathsf{append}(\mathsf{cons}(x_1, \mathsf{nil}), \mathsf{cons}(x_2, \mathsf{nil}))$, we could get the *rewrite sequence* $t \to_{\mathcal{R}} \mathsf{cons}(x_1, \mathsf{append}(\mathsf{nil}, \mathsf{cons}(x_2, \mathsf{nil}))) \to_{\mathcal{R}} \mathsf{cons}(x_1, \mathsf{cons}(x_2, \mathsf{nil}))$. We can make a rewrite step $t \to_{\mathcal{R}} t'$ using a rule $\ell \to r \in \mathcal{R}$ by replacing an instance $\ell\sigma$ of the left-hand side ℓ in t by the corresponding instance $r\sigma$ of the right-hand side r to get t'.

Termination of $\mathcal{R}$ means that every rewrite sequence will end in a term for which no further rewrite step is possible. For term rewriting, the concept of "ranking function" has been adapted via interpretations to (extended) monotone algebras (see, e.g., [31, 43, 10]). A prominent example are *polynomial interpretations* [27], which map function symbols f to polynomials $f_{\mathcal{P}ol}$. This mapping is extended homomorphically to terms (corresponding to the state of the program): $[x]_{\mathcal{P}ol} = x$ for variables x, and $[f(t_1, \ldots, t_n)]_{\mathcal{P}ol} = f_{\mathcal{P}ol}([t_1]_{\mathcal{P}ol}, \ldots, [t_n]_{\mathcal{P}ol})$ for terms $f(t_1, \ldots, t_n)$. To compare two terms s and t, we compare polynomials $[s]_{\mathcal{P}ol} > [t]_{\mathcal{P}ol}$ over $\mathbb{N}$. A term rewrite system $\mathcal{R}$ is terminating if $[\ell]_{\mathcal{P}ol} > [r]_{\mathcal{P}ol}$ holds for all rules $\ell \to r \in \mathcal{R}$. (Interpretations to other monotone algebras are defined analogously.)

▶ **Example 2.** The interpretation $\mathcal{P}ol$ with $\mathsf{append}_{\mathcal{P}ol}(x_1, x_2) = 2 \cdot x_1 + x_2$, $\mathsf{cons}_{\mathcal{P}ol}(x_1, x_2) = x_1 + x_2 + 1$, and $\mathsf{nil}_{\mathcal{P}ol} = 1$ proves termination of $\mathcal{R}$ from Example 1. For rule (1) we get $1 + ys > ys$, and for rule (2) we get $2 \cdot x + 2 \cdot xs + 2 + ys > x + 2 \cdot xs + ys + 1$, which both hold.

To find such interpretations automatically, commonly a *template-based* approach is used. For example, we could use $\mathcal{P}ol$ with parameters a_i whose values are yet to be determined: $\mathsf{append}_{\mathcal{P}ol}(x_1, x_2) = a_1 \cdot x_1 + a_2 \cdot x_2 + a_3$, $\mathsf{cons}_{\mathcal{P}ol}(x_1, x_2) = a_4 \cdot x_1 + a_5 \cdot x_2 + a_6$, and $\mathsf{nil}_{\mathcal{P}ol} = a_7$. From rule (1) we get $[\mathsf{append}(\mathsf{nil}, ys)]_{\mathcal{P}ol} > [ys]_{\mathcal{P}ol}$ and thus $a_1 \cdot a_7 + a_2 \cdot ys + a_3 > ys$. Sound quantifier elimination techniques like the absolute positiveness criterion [22] (note that ys is implicitly universally quantified on $\mathbb{N}$), yield a sufficient condition on our parameters a_i for the earlier inequality: $a_1 \cdot a_7 + a_3 > 0 \wedge a_2 \geq 1$. This constraint is a *quantifier-free formula in non-linear integer arithmetic* (QF_NIA). Although satisfiability for QF_NIA is undecidable [32], existing SMT solvers can be used to find a solution for the a_i and hence a concrete polynomial interpretation $\mathcal{P}ol$ such that $[\mathsf{append}(\mathsf{nil}, ys)]_{\mathcal{P}ol} > [ys]_{\mathcal{P}ol}$.

Several extensions of these polynomial interpretations to $\mathbb{N}$ are also automated via QF_NIA: polynomial interpretations with negative coefficients [20, 21, 12, 19] and with general max and min operators [13], matrix interpretations [10], interpretations to elementary functions and to ordinal functions beyond Peano arithmetic [42], and partly strongly monotone polynomial interpretations for a combination with inductive theorem proving [14].

▶ Remark. Termination proving for term rewriting seems to have been a driving force for the development of SMT solvers for QF_NIA. In many instances of the annual SMT-COMP (`http://www.smt-comp.org`) competition of SMT solvers in recent years, an SMT solver scored highest that was developed as part of or in close connection with a termination prover for term rewriting. In 2009: Barcelogic-QF_NIA [4, 5], using an incomplete reduction from QF_NIA to QF_LIA ("L" for "linear", rendering satisfiability decidable; the termination prover NaTT [40] also uses a related reduction); in 2010: MiniSmt (based on TTT2) [25, 41]; in 2011, 2014, and 2015: AProVE [17, 12], the latter two tools using bit-blasting to SAT.

Moreover, Lucas [30] proposed polynomial interpretations that map to $\mathbb{R}^{\geq 0}$ instead of $\mathbb{N}$. Although the resulting SMT formulas to search for such interpretations are in QF_NRA ("R" for "real"), where satisfiability is decidable [37], for efficiency such constraints are often solved not by decision procedures, but by techniques that search on a finite domain [16, 41].

Floyd [11] also mentions *lexicographic combinations* of ranking functions. For term rewriting, they can be found in a fully modular way, independently for each component, with reduction pairs [26], e.g., by interpretations to extended monotone algebras. Reduction pairs are usually combined with the Dependency Pair framework [2, 18, 21], allowing to lift some monotonicity restrictions and to use proof steps not based on well-founded orders altogether.

Due to the needs of equational theorem proving as an early application, for term rewriting usually termination for *arbitrary* initial terms is considered. There has been little work on proving termination for a *restricted* set of initial terms. Notable exceptions include [9, 23].

3 Imperative Programs

Papers on proving termination of imperative programs are often based on a representation of the program as a transition system on tuples of integer variables, analogous to a rewrite system with built-in integer arithmetic. For instance, the below program (in Python syntax) can be equivalently represented by the transitions (or *constrained* rewrite rules) $\mathcal{T}$ in the box below it.

```
if x >= 0:          # l_0
  while x != 0:     # l_1
    x = x - 1       # l_2
                    # l_3
```

$$\begin{array}{ll} \ell_0(x) \to \ell_1(x) & [x \geq 0] \\ \ell_1(x) \to \ell_2(x) & [x \neq 0] \\ \ell_2(x) \to \ell_1(x-1) & \\ \ell_1(x) \to \ell_3(x) & [x = 0] \end{array}$$

Note that with the notion of termination from the previous section, this transition system $\mathcal{T}$ is non-terminating: $\ell_2(-1) \to_{\mathcal{T}} \ell_1(-2) \to_{\mathcal{T}} \ell_2(-2) \to_{\mathcal{T}} \cdots$ However, this sequence is not *reachable* in the original program. The only allowed initial terms (states) for this program are $\ell_0(z)$ for some integer z. From these states, the program is indeed terminating. Thus, termination provers for imperative programs also need to consider *safety* (unreachability).

To use information from initial states, current techniques find *invariants* (here: $x \geq 0$) either statically in a pre-analysis [34] or dynamically during proof search. Here SMT-based techniques can search both for invariants [8, 6, 28] and for (possibly lexicographic) ranking functions [35, 1]. For instance, Terminator [7] finds linear ranking functions for simple loops [35] proposed by a safety prover as possible counterexamples to termination. These ranking functions are combined to transition invariants [36] or, as in [8], to lex. ranking functions. When the combined ranking functions "cover"

all program executions, termination is proved. Thus, whereas in the DP framework or in the construction of lex. ranking functions, essentially *all* program executions using a certain transition presumably infinitely often must be proved terminating in a *single* proof step, Terminator tries to prove termination just for a *single path* through the program at a time. The approach exploits that a termination argument for some path through the program will often also prove termination of other paths so that often only finitely many executions need to be considered explicitly. A cooperating combination of both approaches is proposed in [6].

4 Concluding Remarks

As this (necessarily incomplete) survey shows, SMT solving nowadays is a prominent approach to automation of termination proving, both for term rewriting and for imperative programs. Thus, improvements to SMT solvers should also lead to improved termination provers.

References

1 C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In *SAS*, 2010.
2 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1–2):133–178, 2000.
3 F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge Univ. Press, 1998.
4 C. Borralleras, S. Lucas, R. Navarro-Marset, E. Rodríguez-Carbonell, and A. Rubio. Solving non-linear polynomial arithmetic via SAT modulo linear arithmetic. In *CADE*, 2009.
5 C. Borralleras, S. Lucas, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. SAT modulo linear arithmetic for solving polynomial constraints. *J. Autom. Reas.*, 48(1):107–131, 2012.
6 M. Brockschmidt, B. Cook, and C. Fuhs. Better termination proving through cooperation. In *CAV*, 2013.
7 B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In *PLDI*, 2006.
8 B. Cook, A. See, and F. Zuleger. Ramsey vs. lexicographic termination proving. In *TACAS*, 2013.
9 J. Endrullis, R. C. de Vrijer, and J. Waldmann. Local termination: theory and practice. *Logical Methods in Computer Science*, 6(3), 2010.
10 J. Endrullis, J. Waldmann, and H. Zantema. Matrix interpretations for proving termination of term rewriting. *J. Autom. Reas.*, 40(2–3):195–220, 2008.
11 R. W. Floyd. Assigning meanings to programs. *Proc. AMS Symposium on Applied Mathematics*, 19:19–31, 1967.
12 C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT solving for termination analysis with polynomial interpretations. In *SAT*, 2007.
13 C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. Maximal termination. In *RTA*, 2008.
14 C. Fuhs, J. Giesl, M. Parting, P. Schneider-Kamp, and S. Swiderski. Proving termination by dependency pairs and inductive theorem proving. *J. Autom. Reas.*, 47(2):133–160, 2011.
15 C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. Proving termination of integer term rewriting. In *RTA*, 2009.
16 C. Fuhs, R. Navarro-Marset, C. Otto, J. Giesl, S. Lucas, and P. Schneider-Kamp. Search techniques for rational polynomial orders. In *AISC*, 2008.
17 J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving termination of programs automatically with AProVE. In *IJCAR*, 2014.

18 J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *J. Autom. Reas.*, 37(3):155–203, 2006.

19 J. Giesl, R. Thiemann, S. Swiderski, and P. Schneider-Kamp. Proving termination by bounded increase. In *CADE*, 2007.

20 N. Hirokawa and A. Middeldorp. Polynomial interpretations with negative coefficients. In *AISC*, 2004.

21 N. Hirokawa and A. Middeldorp. Tyrolean termination tool: Techniques and features. *Information and Computation*, 205(4):474–511, 2007.

22 H. Hong and D. Jakuš. Testing positiveness of polynomials. *J. Autom. Reas.*, 21(1):23–38, 1998.

23 J. Iborra, N. Nishida, and G. Vidal. Goal-directed and relative dependency pairs for proving the termination of narrowing. In *LOPSTR*, 2009.

24 C. Kop and N. Nishida. Term rewriting with logical constraints. In *FroCoS*, 2013.

25 M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean termination tool 2. In *RTA*, 2009.

26 K. Kusakari, M. Nakamura, and Y. Toyama. Argument filtering transformation. In *PPDP*, 1999.

27 D. S. Lankford. Canonical algebraic simplification in computational logic. Technical Report ATP-25, University of Texas, 1975.

28 D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Proving termination of imperative programs using Max-SMT. In *FMCAD*, 2013.

29 S. Lucas. Context-sensitive computations in functional and functional logic programs. *Journal of Functional and Logic Programming*, 1998(1):1–61, 1998.

30 S. Lucas. Polynomials over the reals in proofs of termination: from theory to practice. *RAIRO – Theoretical Informatics and Applications*, 39(3):547–586, 2005.

31 Z. Manna and S. Ness. On the termination of Markov algorithms. In *HICSS*, 1970.

32 Y. Matiyasevich. Enumerable sets are Diophantine. *Soviet Mathematics (Dokladi)*, 11(2):354–357, 1970.

33 R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, 2006.

34 C. Otto, M. Brockschmidt, C. v. Essen, and J. Giesl. Automated termination analysis of Java Bytecode by term rewriting. In *RTA*, 2010.

35 A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *VMCAI*, 2004.

36 A. Podelski and A. Rybalchenko. Transition invariants. In *LICS*, 2004.

37 A. Tarski. *A Decision Method for Elementary Algebra and Geometry.* University of California Press, 1951.

38 A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42(2):230–265, 1936.

39 A. M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, 1949.

40 A. Yamada, K. Kusakari, and T. Sakabe. Nagoya Termination Tool. In *RTA-TLCA*, 2014.

41 H. Zankl and A. Middeldorp. Satisfiability of non-linear (ir)rational arithmetic. In *LPAR (Dakar)*, 2010.

42 H. Zankl, S. Winkler, and A. Middeldorp. Beyond polynomials and Peano arithmetic – automation of elementary and ordinal interpretations. *J. Symb. Comput.*, 69:129–158, 2015.

43 H. Zantema. Termination of term rewriting: Interpretation and type elimination. *J. Symb. Comput.*, 17(1):23–50, 1994.

Symbolic Enumeration of One-Rule String Rewriting Systems

Alfons Geser[1], Johannes Waldmann[2], and Mario Wenzel[3]

1 **Fakultät EIT, HTWK Leipzig, Germany** alfons.geser@htwk-leipzig.de
2 **Fakultät IMN, HTWK Leipzig, Germany** johannes.waldmann@htwk-leipzig.de
3 **Fakultät IMN, HTWK Leipzig, Germany**

Abstract

The purpose of the enumeration of one-rule string rewriting systems is to benchmark methods for proving termination automatically, in particular, to extract interesting cases that merit further attention. We report on a new enumeration approach that represents sets of rewriting systems as the set of models of a binary decision diagram. We relate this to methods and results from the literature, and present preliminary results of experiments.

1 Motivation

Rewriting is a model of computation. The termination status of a rewriting system — does it terminate or not? — is a practically relevant piece of information. Small, hard examples of a restricted shape play a crucial role. They allow to uncover, demonstrate and communicate weaknesses of existing approaches and they drive the invention of new methods. One example of a shape restriction is the restriction to unary symbols which means the switching to string rewriting.

The restriction of size and shape may or may not weaken the descriptive power. E.g., termination is decidable for one-rule string rewriting systems (SRSs) $l \to r$ with $l \in 0^*1^*$ [13] whence it is, particularly, not Turing-complete. On the other hand, termination of one-rule term rewriting is undecidable. And there are one-element bases for combinatory logic, which are Turing-complete. The study of restricted systems per se is justified by finding out the thresholds between these classes.

Small string rewriting termination problems have indeed triggered new approaches.

- The first automated termination proof for Zantema's problem [16] $a^2b^2 \to b^3a^3$ obtained from (RFC) matchbounds [6] was later generalized to term rewriting [8].
- The first termination proof (automated or not) for Zantema's other problem $a^2 \to bc, b^2 \to ac, c^2 \to ab$ by matrix interpretations [7] was also generalized later to term rewriting [2] and to complexity analysis [12].

2 Explicit Enumeration

Small hard examples are found by enumerating all small instances, and filtering out those that are

- **easy**, in the sense that they belong to a class that is known to have a decidable termination problem; or
- **redundant**, in the sense that there is a smaller system that is known to have the same termination status. Here, "smaller" is with respect to a well-founded order that is a refinement of the order by size.

For instance, if $|l| \geq |r|$ then $l \to r$ is easy: it terminates iff $l \neq r$. Or, if there is a bijective renaming ϕ of letters such that $\phi(l) \to \phi(r)$ is lexicographically smaller, then $l \to r$

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 8; pp. 8:1–8:5.

is redundant. A system $l \to r$ is also redundant if there is a bijective renaming ϕ of letters such that $\phi(\tilde{l}) \to \phi(\tilde{r})$ is smaller, where $\tilde{s}$ denotes the reversal of string s. We call $l \to r$ **canonical** if it is not redundant in either of these two ways.

The overhead of an enumeration can be reduced substantially if one avoids some of the systems that are easy or redundant. Kurth [9] enumerates all length-increasing, canonical one-rule SRSs $l \to r$ for $|r| \leq 6$. Geser [4] extends this enumeration to $|r| \leq 9$. Both enumerations follow this approach:

```
foreach System s in canonical_systems { if not (easy (s)) then print (s) }
```

3 Symbolically Representing Sets of Rewriting Systems as BDDs

We present a radically different approach that avoids explicit enumeration: We represent SRSs as models of binary decision diagrams (BDDs [1]). We represent all rules $l \to r$ of a certain shape (fixed length of l and r) and a fixed alphabet as assignments of Boolean variables, using some encoding scheme. We formulate criteria $P_1, P_2, \ldots$ of rewriting systems as Boolean formulas $P_1', P_2', \ldots$ compatible with the chosen encoding.

Instead of explicitly enumerating all $l \to r$ and then checking criteria $P_1, P_2, \ldots$ one after another, we compute the BDD representation P' of $P_1' \wedge P_2' \wedge \ldots$ and then enumerate the models of P':

```
foreach Assignment a in models (P1 and P2 and ...) { print (decode (a)) }
```

Additional advantages of this approach are:

- we can count the number of models without actually enumerating them,
- we can use any Boolean combination of criteria to investigate relations between them, e.g., implications.

4 Criteria related to Termination of Standard Rewriting

The following criteria are used. These are either obvious or well-known, except for (two-letter) coding.

Redundancy criteria:

- $l \to r$ is not canonical. A canonical rule is lexicographically minimal in the equivalence class of rules w.r.t. renaming or reversal.
  - reversal: $ab \to baa$ is transformed to $ba \to aab$
  - renaming: $ab \to baa$ is transformed by $\{a \mapsto b, b \mapsto a\}$ to $ba \to abb$

  The equivalence class of $ab \to baa$, restricted to alphabet $\{a, b\}$ is $\{ab \to baa, ba \to abb, ba \to aab, ab \to bba\}$. The minimal element w.r.t. the order $rl <_{\text{lex}} r'l'$ is $ba \to aab$.
- $l \to r$ is *bordered*, i.e. both l and r begin and end with the same non-empty string [4]. Example: $abba \to abaaba$ is bordered by a, and the termination problem is reduced to $[bb] \to [b][][b]$, over alphabet $\{[], [b], [bb]\}$.
- two-letter-coding. For example, $bca \to aabc$ is reduced to $[bc]a \to aa[bc]$ via the code $\{a, bc\}$, where $[bc]$ is treated as a single letter.

Ease criteria:

- $l \to r$ deletes a letter: $\Sigma(l) \not\subseteq \Sigma(r)$.
- Kurth's Criterion A: a letter occurs more often in l than in r. This class includes the deleting rules.

- Kurth's Criterion D: l is not a factor of r, and either there are no overlaps between the end of l and the begin of r or there are no overlaps between the end of r and the begin of l. Example: $aba \to aaabb$. The end of $aaabb$ has no overlaps with the begin of aba.
- Loops of length one: l is a factor of r.
- Loops of length two (by analysis of overlaps).
- McNaughton's criterion [11]: there exists an inhibitor $i \in \Sigma(r) \setminus \Sigma(l)$.
- Sénizergues' criterion [13]: l has the shape a^*b^*.
- $l \to r$ is grid [5]: there is a letter c with $|l|_c > 0$ and $|l|_c \geq |r|_c$ Example: $bbab \to abbaaabaa$. This class includes the Criterion A rules.

5 Implementation

Our implementation (`https://gitlab.imn.htwk-leipzig.de/waldmann/srs-count`) uses Haskell and the well-known BDD C-library CUDD [14].

We use the "one-hot" encoding for letters where the i-th variable being true means this letter is the i-th letter of Σ while all other variables for that letter are false. A word is a list of letters and a rule is a pair of words. In total, the encoding of $l \to r$ uses $(|l| + |r|) \cdot |\Sigma|$ propositional variables.

Criteria from Section 4 are expressed with the help of predicates for equality and order on letters, for the prefix relation on words, and so on. A consistency predicate expresses the one-hot property. It is always part of the main conjunction. Other predicates, or their negation, can be included via command line arguments. The most expensive criterion is canonicity w.r.t. reversal and renaming, where the number of BDD operations depends exponentially on the size of the alphabet.

The implementation computes the BDD and enumerates its models and decodes them to SRSs. Termination provers `matchbox` [15] and T$_{\mathsf{T}}$T$_2$ [10] can be called for further filtering.

```
srs-count -n True -R True -a True -i False, -g False -o False
  --results 20 --matchbox no 3 6 9
```

This example call computes the first 20 systems with a left-hand side of size 6, a right-hand side of size 9 and a size-3 alphabet that are canonical by re(n)aming and (R)eversal-and-(R)enaming, use (a)ll 3 letters of the alphabet, do not have an (i)nhibitor, are not a (g)rid-rule and do not have a loop of length (o)ne, while `matchbox` still has a non-termination proof.

Additionally, we allow the enumeration to be split or restricted using patterns (globs) like `-globleft="ab*"`, which would restrict the left-hand side to words of the language $a \cdot b \cdot \Sigma^*$. This replaces Boolean variables by constants, and makes for smaller BDDs. For a complete enumeration, we apply several such patterns to distribute the computation across multiple computers.

6 Results

We confirmed that symbolic and explicit enumeration agree for $|r| \leq 9$. Table 1 shows the numbers obtained by an explicit enumeration, using Geser's original implementation, of all length-increasing, canonical one-rule SRSs ("all"), and of those SRSs that satisfy both $|l| \geq |\Sigma|$ and $|r| \geq |l| + |\Sigma|$ ("restricted"). The number of non-grid, non-inhibitor systems, obtained through filtering, is the same in both cases. Further filtering out 1-loop and Criterion D yields the next column. The final column shows the number after further filtering out 2-loop and bordered ("fast check criteria"). The table illustrates that the explicit

generate-and-filter approach quickly becomes prohibitively expensive and less useful since the share of interesting systems becomes smaller as the system size grows.

Using symbolic enumeration, we were able to reproduce the results from the second-to-last column up to $|r| \leq 8$ in less than 10 seconds ($|r| \leq 9$ in 3½ minutes) on a 3.2 GHz processor.

$\|r\|$	all	restricted	non-grid, non-inhibitor	..., non-1-loop, non-crit-D	non-fast-criteria
2	2	1	0	0	0
3	21	2	2	0	0
4	226	20	8	1	0
5	3 929	103	30	7	4
6	96 029	1 699	207	68	45
7	3 151 054	18 345	1 618	540	440
8	130 792 338	396 184	16 594	4 994	4 265
9	6 641 134 837	6 642 933	196 476	49 814	43 535
10	?	173 514 078	2 710 745	562 258	493 855
11	?	4 039 563 892	42 735 641	7 213 316	6 346 721

Table 1 Numbers of length-increasing, canonical one-rule SRSs

In order to obtain fresh hard termination problems, we have enumerated and filtered all one-rule SRS with $|r| \leq 14$ and $|\Sigma|$ no larger than 3, using all stated criteria except criterion D (which was a recent addition to our implementation).

This left about $7.66 \cdot 10^9$ systems, which we have filtered using `matchbox` [15], applying only RFC match bounds for termination, and forward closure enumeration for non-termination, and spending no more than 1 second per problem (on our machines). Enumeration and filtering took 30.000 CPU hours, approximately.

This left 671 systems, on which we ran T$_{\text{T}}$T$_2$ [10] and AProVE [3] on starexec, using 300 seconds as a timeout. We obtained 226 systems where termination currently cannot be shown automatically, and which we will submit for TPBD. Four random examples are:

$$aabaaaa \to aaaaaabaab, \qquad babbaabba \to abbaabbabba,$$
$$bababababaa \to aababababababa, \qquad cabababa \to abababab\text{ccccca}.$$

7 Extension to Termination of Cycle Rewriting

Recently, there has been an interest in cycle rewriting [17]. A string rewriting system R over Σ defines a *cycle rewriting relation* $\overset{\circ}{\to}_R$ on Σ^* that is the composition of the standard *conjugacy* relation $uv \equiv vu$ with the standard rewrite relation $\to_R$.

Our approach for symbolically enumerating interesting one-rule rewriting systems is easily applicable for cycle rewriting, and in fact we simply use our existing implementation, and switch off a few criteria. From the list of properties in Section 4, we omit the following because their applicability needs further research: Kurth's criterion D, the grid criterion and Sénizergues' criterion. Note that we can use the inhibitor criterion for reduction: If R has an inhibitor, then cycle termination of R is equivalent to standard termination of R, which is (in that case) decidable.

For cycle termination, there was no previous enumeration. We generated $3.1 \cdot 10^6$ length-increasing systems without the applicable properties with $|r| \leq 9$ and $|\Sigma|$ no larger than 3. The initial generation took 5½ minutes on a 2.1 GHz i3 processor.

These are the 6 smallest one-rule SRSs for which matchbox could not determine the status of cycle termination:

$$baba \to abaaabab, \qquad ababba \to aabbabab, \qquad abaaba \to aababaab,$$
$$baba \to abaaaabab, \qquad baabba \to aabbaaabb, \qquad ababbab \to abbababba.$$

References

1. Sheldon B. Akers. Binary Decision Diagrams. *IEEE Trans. Computers*, 27(6):509–516, 1978.
2. Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix Interpretations for Proving Termination of Term Rewriting. *J. Autom. Reasoning*, 40(2-3):195–220, 2008.
3. Jürgen Giesl et al. Automated Program Verification Environment. `http://aprove.informatik.rwth-aachen.de/`, 2016.
4. Alfons Geser. *Is Termination Decidable for String Rewriting With Only One Rule*. Habilitationsschrift, Universität Tübingen, 2001.
5. Alfons Geser. Decidability of Termination of Grid String Rewriting Rules. *SIAM J. Comput.*, 31(4):1156–1168, 2002.
6. Alfons Geser, Dieter Hofbauer, and Johannes Waldmann. Match-Bounded String Rewriting Systems. *Appl. Algebra Eng. Commun. Comput.*, 15(3-4):149–171, 2004.
7. Dieter Hofbauer and Johannes Waldmann. Termination of String Rewriting with Matrix Interpretations. In Frank Pfenning, editor, *RTA 2006*, volume 4098 of *LNCS*, pages 328–342. Springer, 2006.
8. Martin Korp and Aart Middeldorp. Match-bounds revisited. *Inf. Comput.*, 207(11):1259–1283, 2009.
9. Winfried Kurth. *Termination und Konfluenz von Semi-Thue-Systemen mit nur einer Regel*. Dissertation, Technische Universität Clausthal, 1990.
10. Harald Zankl Martin Korp, Christian Sternagel and Aart Middeldorp. Tyrolean Termination Tool 2. `http://cl-informatik.uibk.ac.at/software/ttt2/`, 2014.
11. Robert McNaughton. Semi-Thue Systems with an Inhibitor. *J. Autom. Reasoning*, 26:409–431, 1997.
12. Georg Moser, Andreas Schnabl, and Johannes Waldmann. Complexity Analysis of Term Rewriting Based on Matrix and Context Dependent Interpretations. In Ramesh Hariharan, Madhavan Mukund, and V. Vinay, editors, *FSTTCS 2008*, volume 2 of *LIPIcs*, pages 304–315. Schloss Dagstuhl - LZI, 2008.
13. Géraud Sénizergues. On the Termination Problem for One-Rule Semi-Thue System. In Harald Ganzinger, editor, *RTA-96*, volume 1103 of *LNCS*, pages 302–316. Springer, 1996.
14. Fabio Somenzi. CUDD: CU Decision Diagram Package Release 3.0.0. `http://vlsi.colorado.edu/~fabio/CUDD`, 2015.
15. Johannes Waldmann. Pure Matchbox. `https://gitlab.imn.htwk-leipzig.de/waldmann/pure-matchbox`, 2016.
16. Hans Zantema and Alfons Geser. A Complete Characterization of Termination of $0^p1^q \to 1^r0^s$. *Appl. Algebra Eng. Commun. Comput.*, 11(1):1–25, 2000.
17. Hans Zantema, Barbara König, and H. J. Sander Bruggink. Termination of Cycle Rewriting. In Gilles Dowek, editor, *RTA-TLCA 2014*, volume 8560 of *LNCS*, pages 476–490. Springer, 2014.

Termination Analysis of Programs with Bitvector Arithmetic by Symbolic Execution*

Jera Hensel, Jürgen Giesl, Florian Frohn, and Thomas Ströder

LuFG Informatik 2, RWTH Aachen University, Germany
{hensel,giesl,florian.frohn,stroeder}@informatik.rwth-aachen.de

Abstract

We recently developed an approach for automated termination analysis of C programs with explicit pointer arithmetic, which is based on symbolic execution [11]. However, similar to many other termination techniques, this approach assumed the program variables to range over mathematical integers instead of bitvectors. This eases mathematical reasoning but is unsound in general. In this paper, we extend our approach in order to handle fixed-width bitvector integers. Thus, we present the first technique for termination analysis of C programs that covers both byte-accurate pointer arithmetic and bit-precise modeling of integers. We implemented our approach in the automated termination prover AProVE [6] and evaluate its power by extensive experiments.

1998 ACM Subject Classification D.2.4 - Software/Program Verification, E.2 - Data Storage Representations, F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Termination, Bitvectors, C Programs, LLVM, Symbolic Execution

1 Introduction

In general, it is unsound to assume mathematical integers in programming languages: The function `f` below does not terminate if `x` has the maximum value of its type. But we can falsely prove termination if we treat `x` and `j` as mathematical integers. For `g`, we could falsely conclude non-termination, although `g` terminates due to the wrap-around for overflows.

```
void f(unsigned int x)  {          void g(unsigned int j) {
    unsigned int j = 0;                while (j > 0) j++; }
    while (j <= x) j++; }
```

In this paper, we adapt our approach for termination of C [11] to the bitvector semantics. To avoid dealing with the intricacies of C, we analyze programs in the intermediate representation of the LLVM compilation framework [9]. Our approach first constructs a *symbolic execution graph* that over-approximates all possible program runs (Sect. 2 and 3). This graph is also used to prove that the program does not result in undefined behavior. In a second step (Sect. 4), this graph is transformed into an *integer transition system (ITS)*, whose termination can be proved by existing techniques. The full version of this paper appeared in [8].

2 LLVM States for Symbolic Execution

In the LLVM code for `g` obtained with the Clang compiler (see Page 2), `j` has type `i32`, as it is a bitvector of length 32. The program has the *basic blocks* `entry`, `cmp`, `body`, and `done`.

In our abstract domain, an LLVM *state* has the form (p, LV, KB, AL, PT). The *program position* p is a pair $(\mathtt{b}, k)$. Here, `b` is the name of the current basic block and k is the index of the next instruction. $LV : \mathcal{V}_\mathcal{P} \to \mathcal{V}_{sym}$ is an assignment to the *local program variables* $\mathcal{V}_\mathcal{P}$ (e.g., $\mathcal{V}_\mathcal{P} = \{\mathtt{j}, \mathtt{ad}, \ldots\}$), where $\mathcal{V}_{sym}$ is an infinite set of symbolic variables.

* Supported by the DFG grant GI 274/6-1.

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 9; pp. 9:1–9:5.

The *knowledge base* KB consists of first-order integer formulas. For concrete states, KB uniquely determines the values of symbolic variables. For abstract states several values are possible.

The *allocation list* AL contains expressions $[\![v_1, v_2]\!]$ for $v_1, v_2 \in \mathcal{V}_{sym}$, which indicate that $v_1 \leq v_2$ and that all addresses between v_1 and v_2 were allocated by an `alloca` instruction.

```
define i32 @g(i32 j) {
entry: 0: ad = alloca i32
       1: store i32 j, i32* ad
       2: br label cmp
cmp:   0: j1 = load i32* ad
       1: j1pos = icmp ugt i32 j1, 0
       2: br i1 j1pos, label body, label done
body:  0: j2 = load i32* ad
       1: inc = add i32 j2, 1
       2: store i32 inc, i32* ad
       3: br label cmp
done:  0: ret void }
```

The fifth component PT is a set of "points-to" atoms $v_1 \hookrightarrow_{\mathtt{ty},i} v_2$ where $v_1, v_2 \in \mathcal{V}_{sym}$, `ty` is an LLVM type, and $i \in \{u, s\}$. This means that the value v_2 of type `ty` is stored at the address v_1, where $i \in \{u, s\}$ indicates whether v_2 represents this value as an unsigned or signed integer. As each memory cell stores one byte, $v_1 \hookrightarrow_{\mathtt{i32},i} v_2$ states that v_2 is stored in the four cells $v_1, \ldots, v_1 + 3$. Finally, we use a state ERR to be reached if we cannot prove absence of undefined behavior (e.g., for non-allowed overflow or a violation of memory safety).

We often identify the mapping LV with the equations $\{\mathtt{x} = LV(\mathtt{x}) \mid \mathtt{x} \in \mathcal{V}_\mathcal{P}\}$. Consider the following abstract state for `g`. It represents states in the `entry` block immediately before executing the instruction in Line 2. Here, $LV(\mathtt{j}) = v_\mathtt{j}$, the memory cells between $LV(\mathtt{ad}) = v_\mathtt{ad}$ and $v_{end} = v_\mathtt{ad} + 3$ have been allocated, and $v_\mathtt{j}$ is stored in the 4 cells $v_\mathtt{ad}, \ldots, v_{end}$.

$$(\ (\mathtt{entry}, 2),\ \{\mathtt{j} = v_\mathtt{j}, \mathtt{ad} = v_\mathtt{ad}\},\ \{v_{end} = v_\mathtt{ad} + 3\},\ \{[\![v_\mathtt{ad}, v_{end}]\!]\},\ \{v_\mathtt{ad} \hookrightarrow_{\mathtt{i32},u} v_\mathtt{j}\}\ ) \tag{1}$$

To construct symbolic execution graphs, for any state a we use a first-order formula $\langle a \rangle$, which contains KB and obvious consequences of AL and PT. If c is a *concrete* state, then for all $v \in \mathcal{V}_{sym}(c)$ there is an $n \in \mathbb{Z}$ with $\models \langle c \rangle \Rightarrow v = n$.

In [11], we used *separation logic* to define formally which concrete states are represented by an abstract state a. For example, the abstract state (1) represents all concrete states $c = ((\mathtt{entry}, 2), LV, KB, AL, PT)$ where the 32-bit integer `j` is stored at the address `ad`.

3 From LLVM to Symbolic Execution Graphs

We now show how to automatically generate a *symbolic execution graph* that over-approximates all runs of a program. We developed symbolic execution rules for all LLVM instructions that are affected by the adaption to bitvectors. Our approach starts with the initial states that one wants to analyze for termination, e.g., with the abstract state A where `j` has an unknown value. In the symbolic execution graph for `g` in Fig. 1, we wrote $\hookrightarrow_{\mathtt{i32}}$ and umax instead of $\hookrightarrow_{\mathtt{i32},u}$ and umax_{32} (where $\mathsf{umax}_n = 2^n - 1$ is the largest unsigned integer with n bits).

The function `g` allocates $[\![v_\mathtt{ad}, v_{end}]\!]$ and stores the value $v_\mathtt{j}$ of `j` at address `ad`. Next, we jump to the block `cmp`. After loading $v_\mathtt{j}$ (stored at the address `ad`) to the program variable `j1`, in E we check whether `j1`'s value in unsigned interpretation is greater than 0 (`icmp ugt`).

We partition the program variables $\mathcal{V}_\mathcal{P}$ into two disjoint sets $\mathcal{U}_\mathcal{P}$ and $\mathcal{S}_\mathcal{P}$. If $\mathtt{x} \in \mathcal{U}_\mathcal{P}$ (resp. $\mathtt{x} \in \mathcal{S}_\mathcal{P}$), then $LV(\mathtt{x})$ is `x`'s value as an unsigned (resp. signed) integer. This is advantageous for instructions like `icmp ugt` and `sgt`, since the LLVM types do not distinguish between unsigned and signed integers. Instead, some LLVM instructions consider their arguments as "unsigned" resp. "signed". We use a heuristic to determine $\mathcal{U}_\mathcal{P}$ and $\mathcal{S}_\mathcal{P}$. It ensures that in each instruction in $\mathcal{P}$, all occurring program variables of type `i`n with $n > 1$ are either from $\mathcal{U}_\mathcal{P}$ or from $\mathcal{S}_\mathcal{P}$. In our example, we obtain $\mathcal{U}_\mathcal{P} = \mathcal{V}_\mathcal{P} = \{\mathtt{j}, \mathtt{ad}, \ldots, \mathtt{inc}\}$ and $\mathcal{S}_\mathcal{P} = \varnothing$. For any $t \in \mathcal{V}_\mathcal{P} \uplus \mathbb{Z}$, let $LV_{u,n}(t)$ represent t as an unsigned integer with n bits: $LV_{u,n}(t) = LV(t)$ if $t \in \mathcal{U}_\mathcal{P}$, $LV_{u,n}(t) = LV(t) \bmod 2^n$ if $t \in \mathcal{S}_\mathcal{P}$, and $LV_{u,n}(t) = t \bmod 2^n$ if $t \in \mathbb{Z}$. $LV_{s,n}(t)$ is defined analogously. Then the following rule evaluates `icmp ugt` symbolically.

In our rules, "p: *ins*" states that *ins* is the instruction at position p. Let a always denote the abstract state *before* the execution step (i.e., above the horizontal line of the rule).

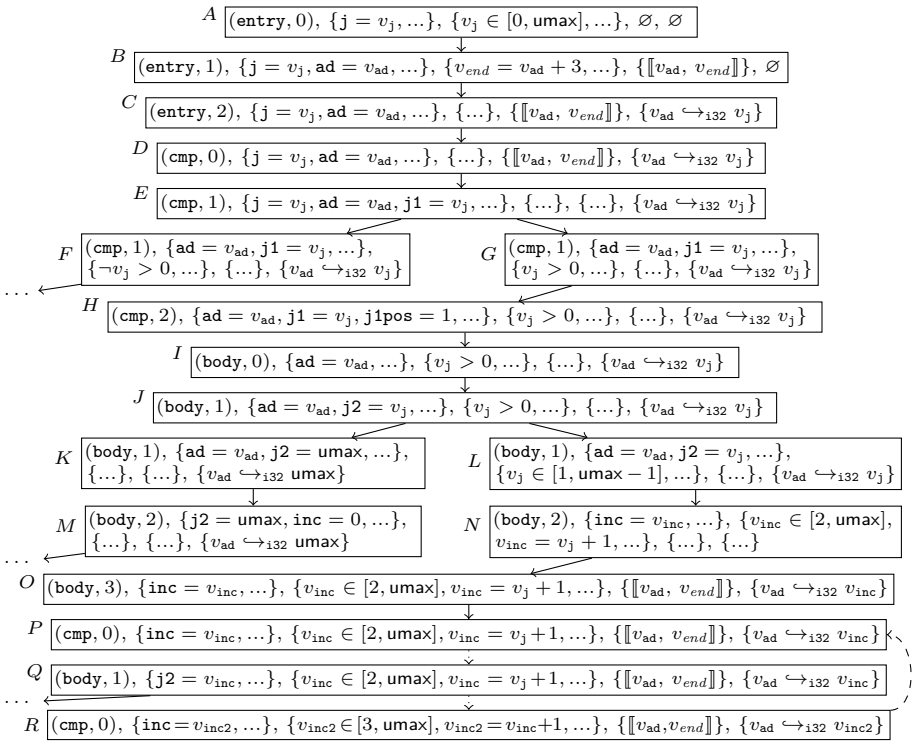


Figure 1 Symbolic execution graph for the function `g`

Moreover, $LV[\mathtt{x} := v]$ is the function where $(LV[\mathtt{x} := v])(\mathtt{x}) = v$ and $(LV[\mathtt{x} := v])(\mathtt{y}) = LV(\mathtt{y})$ for $\mathtt{y} \neq \mathtt{x}$. If $p = (\mathtt{b}, k)$, then $p^+ = (\mathtt{b}, k+1)$ is the position of the next instruction in the same block. Finally, $size(\mathtt{ty})$ is the number of bits required for values of type `ty`.

`icmp ugt` (p: **"`x = icmp ugt ty` t_1, t_2" with** $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\mathtt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)}$$ if $v \in \mathcal{V}_{sym}$ is fresh and if

either $\models \langle a \rangle \Rightarrow (LV_{u,size(\mathtt{ty})}(t_1) > LV_{u,size(\mathtt{ty})}(t_2))$ and φ is "$v = 1$"
or $\models \langle a \rangle \Rightarrow (LV_{u,size(\mathtt{ty})}(t_1) \leq LV_{u,size(\mathtt{ty})}(t_2))$ and φ is "$v = 0$"

However, in our example the value of $LV_{u,32}(\mathtt{j1}) = LV(\mathtt{j1}) = v_\mathtt{j}$ is unknown. Therefore, we first have to *refine* State E to States F and G such that the comparison can be decided. For this case analysis, we use the following rule. The rules for other comparisons are analogous.

`icmp ugt` refinement (p: **"`x = icmp ugt ty` t_1, t_2" with** $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p, LV, KB \cup \{\varphi\}, AL, PT) \quad | \quad (p, LV, KB \cup \{\neg\varphi\}, AL, PT)}$$ if

φ is "$LV_{u,size(\mathtt{ty})}(t_1) > LV_{u,size(\mathtt{ty})}(t_2)$" and we have both $\not\models \langle a \rangle \Rightarrow \varphi$ and $\not\models \langle a \rangle \Rightarrow \neg\varphi$

If $\neg v_\mathtt{j} > 0$ (State F), we `return`. If $v_\mathtt{j} > 0$ (State G), the `branch` instruction leads us to the `body` block. In the step from State I to J, again the value $v_\mathtt{j}$ stored at $v_\mathtt{ad}$ is loaded to `j2`. The next instruction is an overflow-sensitive addition. If $v_\mathtt{j} < \mathsf{umax}_{32}$, then $v_\mathtt{j} + 1$ is assigned to `inc`. But if $v_\mathtt{j} = \mathsf{umax}_{32}$, then there is an overflow. If KB does not contain enough information to decide whether an overflow occurs, we perform a case analysis.

unsigned `add` refinement (p: **"`x = add in` t_1, t_2" with** $\mathtt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p, LV, KB \cup \{\varphi\}, AL, PT) \quad | \quad (p, LV, KB \cup \{\neg\varphi\}, AL, PT)}$$ if $\mathtt{x} \in \mathcal{U}_\mathcal{P}$ and

φ is "$LV_{u,n}(t_1) + LV_{u,n}(t_2) \leq \mathsf{umax}_n$", where $\not\models \langle a \rangle \Rightarrow \varphi$ and $\not\models \langle a \rangle \Rightarrow \neg\varphi$

Therefore, State J is refined to K and L. If no overflow can occur, then the result is the

addition of the operators. Thus, State L evaluates to N, where $v_{\texttt{inc}} = v_{\texttt{j}} + 1$.

unsigned `add` without overflow (p: "`x = add i`n t_1, t_2" **with** $\texttt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\texttt{x} := v],\ KB \cup \{\varphi\},\ AL,\ PT)} \quad \text{if } v \in \mathcal{V}_{sym} \text{ is fresh, } \texttt{x} \in \mathcal{U}_\mathcal{P},$$

$\models \langle a \rangle \Rightarrow (LV_{u,n}(t_1) + LV_{u,n}(t_2) \in [0, \mathsf{umax}_n])$, and φ is "$v = LV_{u,n}(t_1) + LV_{u,n}(t_2)$"

For an overflow, due to the wrap-around, the unsigned result is the sum of the operands minus the type size 2^n. So in the evaluation of K to M, we have $v_{\texttt{inc}} = \mathsf{umax}_{32} + 1 - 2^{32} = 0$.

`add` with overflow (p: "`x = add i`n t_1, t_2" **with** $\texttt{x} \in \mathcal{V}_\mathcal{P}$, $t_1, t_2 \in \mathcal{V}_\mathcal{P} \cup \mathbb{Z}$)

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p^+,\ LV[\texttt{x} := v],\ KB \cup \{v = LV_{u,n}(t_1) + LV_{u,n}(t_2) - 2^n\},\ AL,\ PT)} \quad \text{if}$$

$\texttt{x} \in \mathcal{U}_\mathcal{P}$, $v \in \mathcal{V}_{sym}$ is fresh, and $\models \langle a \rangle \Rightarrow (LV_{u,n}(t_1) + LV_{u,n}(t_2) > \mathsf{umax}_n)$

For M, the execution ends after some steps. For N, after storing $v_{\texttt{inc}}$ to $v_{\texttt{ad}}$, we branch to block `cmp` again. State P is similar to D. So we continue the execution in P, where the steps from P to Q are similar to the steps from D to J. Q is again refined and in the case where no overflow occurs, we finally reach State R at the same program position as D and P.

To obtain *finite* symbolic execution graphs, we can *generalize* states whenever an evaluation visits a program position $(\texttt{b}, k)$ multiple times. We say that a' is a *generalization* of a with the instantiation μ whenever the conditions (b) – (e) of the following rule from [11] are satisfied. Again, a is the state *before* the generalization step and a' is the state *resulting* from the generalization. See [11] for a heuristic to compute suitable generalizations automatically.

generalization with μ

$$\frac{(p,\ LV,\ KB,\ AL,\ PT)}{(p',\ LV',\ KB',\ AL',\ PT')} \quad \text{if}$$

(a) a has an incoming "evaluation edge" (not just refinement or generalization edges)
(b) $LV(\texttt{x}) = \mu(LV'(\texttt{x}))$ for all $\texttt{x} \in \mathcal{V}_\mathcal{P}$
(c) $\models \langle a \rangle \Rightarrow \mu(KB')$
(d) if $[\![v_1, v_2]\!] \in AL'$, then $[\![\mu(v_1), \mu(v_2)]\!] \in AL$
(e) for $i \in \{u, s\}$, if $(v_1 \hookrightarrow_{\texttt{ty},i} v_2) \in PT'$, then $(\mu(v_1) \hookrightarrow_{\texttt{ty},i} \mu(v_2)) \in PT$

In our graph, P is a generalization of State R using an instantiation μ with $\mu(v_{\texttt{j}}) = v_{\texttt{inc}}$ and $\mu(v_{\texttt{inc}}) = v_{\texttt{inc2}}$. So we can conclude the graph construction with a (dashed) *generalization edge* from R to P. A symbolic execution graph is *complete* if all its leaves correspond to `ret` instructions (so in particular, the graph does not contain ERR). So a program with a complete symbolic execution graph as in Fig. 1 does not exhibit undefined behavior.

When representing bitvectors by relations on $\mathbb{Z}$, the wrap-around for overflows can either be handled by case analysis or by "modulo" relations. We use a hybrid approach with case analysis for instructions like addition and with "modulo" for operations like multiplication. We refer to [8] for details on the handling of further LLVM instructions whose symbolic execution rules have to be adapted to bitvector arithmetic.

4 From Symbolic Execution Graphs to Integer Systems

After the graph construction, we extract an *integer transition system* (ITS) from the cycles of the symbolic execution graph and use existing techniques (e.g., [10]) to prove its termination.

ITSs can be represented as graphs whose nodes correspond to program locations and whose edges correspond to transitions. A transition is labeled with conditions required for its application. These conditions are formulas over a set of variables $\mathcal{V}$ and a set $\mathcal{V}' = \{x' \mid x \in \mathcal{V}\}$ which refers to the values of the variables *after* applying the transition.

The only cycle of the symbolic execution graph in Fig. 1 is the one from P to R and back. The resulting ITS is shown in Fig. 2. The values of the variables do not change in transitions that correspond to evaluation edges of the symbolic execution graph. For the generalization edge from R to P with the instantiation μ, the corresponding transition in the ITS gets the condition $v' = \mu(v)$ for all $v \in \mathcal{V}_{sym}(P)$. So we obtain the condition $v'_{\texttt{inc}} = \mu(v_{\texttt{inc}})$, i.e., $v'_{\texttt{inc}} = v_{\texttt{inc2}} = v_{\texttt{inc}} + 1$. In contrast, $v_{\texttt{inc2}}$'s value can change arbitrarily here, since $v_{\texttt{inc2}} \notin \mathcal{V}_{sym}(P)$. Moreover, the transitions of the ITS contain conditions like $v_{\texttt{inc}} \leq \mathsf{umax}_{32}$, which are also present in the states P – R. Termination of this ITS can easily be proved by standard techniques. In [8, 11] we show that termination of the ITS implies termination of the analyzed LLVM program.

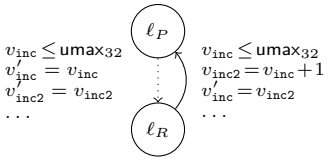


Figure 2 ITS for function g

5 Conclusion

We adapted our approach for proving termination of C (resp. LLVM) programs to bitvectors. Our approach was implemented in AProVE [6], which won the *SV-COMP* 2015 and 2016 competitions at *TACAS* for termination of C programs.[1] Since we represent bitvectors by relations on $\mathbb{Z}$, we can use standard SMT solving on $\mathbb{Z}$ and standard termination analysis on $\mathbb{Z}$ for the symbolic execution and the termination proofs in our approach.

There are few other methods and tools for termination of bitvector programs (e.g., KITTeL [5], TAN [3], 2LS [2], Juggernaut [4], Ultimate [7]). The full version of our paper [8] (which contains a theoretical and experimental comparison with related work), our implementation, and further symbolic execution rules are available online [1].

References

1 AProVE: `http://aprove.informatik.rwth-aachen.de/eval/Bitvectors/`.
2 H.-Y. Chen, C. David, D. Kroening, P. Schrammel, and B. Wächter. Synthesising interprocedural bit-precise termination proofs. In *Proc. ASE '15*, pages 53–64, 2015.
3 B. Cook, D. Kroening, P. Rümmer, and C. Wintersteiger. Ranking function synthesis for bit-vector relations. *Formal Methods in System Design*, 43(1):93–120, 2013.
4 C. David, D. Kroening, and M. Lewis. Unrestricted termination and non-termination arguments for bit-vector programs. In *Proc. ESOP '15*, LNCS 9032, pages 183–204, 2015.
5 S. Falke, D. Kapur, and C. Sinz. Termination analysis of imperative programs using bitvector arithmetic. In *Proc. VSTTE '12*, LNCS 7152, pages 261–277, 2012.
6 J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving termination of programs automatically with AProVE. In *Proc. IJCAR '14*, LNAI 8562, pages 184–191, 2014.
7 M. Heizmann, J. Hoenicke, J. Leike, and A. Podelski. Linear ranking for linear lasso programs. In *Proc. ATVA '13*, LNCS 8172, pages 365–380, 2013.
8 J. Hensel, J. Giesl, F. Frohn, and T. Ströder. Proving termination of programs with bitvector arithmetic by symbolic execution. *Proc. SEFM '16*, LNCS 9763, pages 234–252, 2016.
9 C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. CGO '04*, pages 75–88, 2004.
10 A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Proc. VMCAI '04*, LNCS 2937, pages 239–251, 2004.
11 T. Ströder, J. Giesl, M. Brockschmidt, F. Frohn, C. Fuhs, J. Hensel, P. Schneider-Kamp, and C. Aschermann. Proving termination and memory safety for programs with pointer arithmetic. In *Proc. IJCAR '14*, LNAI 8562, pages 208–223, 2014.

[1] See `http://sv-comp.sosy-lab.org/`.

Non-deterministic Characterisations*

Cynthia Kop

Department of Computer Science, University of Copenhagen (DIKU)
kop@di.ku.dk

Abstract

In this paper, we extend Jones' result—that cons-free programming with k^{th}-order data and a call-by-value strategy characterises $\texttt{EXP}^k\texttt{TIME}$—to a more general setting, including pattern-matching and non-deterministic choice. We show that the addition of non-determinism is unexpectedly powerful in the higher-order setting. Nevertheless, we can obtain a non-deterministic parallel to Jones' hierarchy result by appropriate restricting rule formation.

1 Introduction

In [4], Jones introduces *cons-free programming*. Working with a small functional programming language, cons-free programs are defined to be *read-only*: recursive data cannot be created or altered (beyond taking sub-expressions), only read from the input. By imposing further restrictions on data order and recursion style, classes of cons-free programs turn out to characterise various deterministic classes in the time and space hierarchies of computational complexity. Most relevantly to this work, cons-free programs with data order k characterise the class $\texttt{EXP}^k\texttt{TIME}$ of decision problems decidable in $\mathcal{O}(\exp_2^k(a \cdot n^b))$ on a Turing Machine.

The classes thus characterised are all *deterministic*: they concern the time and space to solve decision problems on a deterministic Turing Machine. As the language considered by Jones is deterministic, a natural question is whether adding non-deterministic choice to the language would increase expressivity accordingly. The answer, at least in the base case, is *no*: following an early result by Cook [2], Bonfante shows [1] that adding a non-deterministic choice operator to cons-free programs with data order 0 makes no difference in expressivity: whether with or without non-deterministic choice, such programs characterise $\texttt{P}$.

In this paper, we consider the generalisation of this question: does adding non-deterministic choice give more expressivity when data of order greater than 0 is admitted? Surprisingly, the answer is yes! However, we do not obtain the non-deterministic classes; rather, non-deterministic cons-free programs of any data order ≥ 1 characterise $\texttt{ELEMENTARY}$, the class $\texttt{EXP}^0\texttt{TIME} \cup \texttt{EXP}^1\texttt{TIME} \cup \texttt{EXP}^2\texttt{TIME} \cup \ldots$. As this is less useful for complexity arguments, we amend cons-freeness with a further restriction—unary variables—which allows us to obtain the expected generalisation: that (thus restricted) cons-free programs of data order k characterise $\texttt{EXP}^k\texttt{TIME}$, whether or not non-deterministic choice is allowed.

We also generalise Jones' language with pattern matching and user-defined constructors.

2 Cons-free programming

For greater generality—and greater ease of expressing examples—we extend Jones' language to a limited functional programming language with pattern matching. We will use terminology from the term rewriting world, but very little of the possibilities of this world.

* Supported by the Marie Skłodowska-Curie action "HORIP", program H2020-MSCA-IF-2014, 658162.

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 10; pp. 10:1–10:5.

2.1 Higher-order Programs

We consider programs using simple types, including product types. The *type order* $o(\sigma)$ of a type σ is defined as follows: $o(\kappa) = 0$ for κ a *sort* (base type), $o(\sigma \times \tau) = \max(o(\sigma), o(\tau))$ and $o(\sigma \Rightarrow \tau) = \max(o(\sigma) + 1, o(\tau))$.

Assume given three disjoint set of identifiers: $\mathcal{C}$ of *constructors*, $\mathcal{D}$ of *defined symbols* and $\mathcal{V}$ of *variables*; each symbol is equipped with a type. Following Jones, we limit interest to constructors with a type $\iota_1 \Rightarrow \ldots \Rightarrow \iota_m \Rightarrow \kappa$ where all ι_i are types of order 0 and κ is a *sort*. *Terms* are expressions s such that $s : \sigma$ can be derived for some type σ using the clauses:

- $c\ s_1 \cdots s_m : \kappa$ if $c : \iota_1 \Rightarrow \ldots \Rightarrow \iota_m \Rightarrow \kappa \in \mathcal{C}$ and each $s_i : \iota_i$
- $a\ s_1 \cdots s_n : \tau$ if $a : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_n \Rightarrow \tau \in \mathcal{V} \cup \mathcal{D}$ and each $s_i : \sigma_i$
- $(s, t) : \sigma \times \tau$ if $s : \sigma$ and $t : \tau$

Thus, constructors cannot be partially applied, while variables and defined symbols can be. If $s : \sigma$, we say σ is the type of s, and let $Var(s)$ be the set of variables occurring in s. A term s is *ground* if $Var(s) = \emptyset$. We say t is a subterm of s, notation $s \trianglerighteq t$, if either $s = t$ or $s = a\ s_1 \cdots s_n$ with $a \in \mathcal{C} \cup \mathcal{F} \cup \mathcal{V}$ and $s_i \trianglerighteq t$ for some i, or $s = (s_1, s_n)$ and $s_i \trianglerighteq t$ for some i. Note that the head of an application is *not* a subterm of the application.

A *rule* is a pair of terms $f\ \ell_1 \cdots \ell_k \to r$ such that (a) $f \in \mathcal{D}$, (b) no defined symbols occur in any ℓ_i, (c) no variable occurs more than once in $f\ \ell_1 \cdots \ell_k$, (d) $Var(r) \subseteq Var(f\ \ell_1 \cdots \ell_k)$, and (e) r has the same type as $f\ \ell_1 \cdots \ell_k$. A *substitution* γ is a mapping from variables to ground terms of the same type, and $s\gamma$ is obtained by replacing variables x in s by $\gamma(x)$.

We fix a set $\mathcal{R}$ of rules, which are *consistent*: if $f\ \ell_1 \cdots \ell_k \to r$ and $f\ q_1 \cdots q_n \to s$ are both in $\mathcal{R}$, then $k = n$; we call k the *arity* of f. The set $\mathcal{DA}$ of *data terms* consists of all ground constructor terms. The set $\mathcal{VA}$ of *values* is given by: (a) all data terms are values, (b) if v, w are values, then (v, w) is a value, (c) if $f \in \mathcal{D}$ has arity k, $n < k$ and $s_1, \ldots, s_n$ are values, then $f\ s_1 \cdots s_n$ is a value if it is well-typed. Note that values whose type is a sort are data terms. The *call-by-value* reduction relation on ground terms is defined by:

- $(s, t) \to^* (v, w)$ if $s \to^* v$ and $t \to^* w$
- $a\ s_1 \cdots s_n \to^* a\ v_1 \cdots v_n$ if each $s_i \to^* v_i$ and either $a \in \mathcal{C}$, or $a \in \mathcal{D}$ and $n < arity(a)$
- $f\ s_1 \cdots s_m \to^* w$ if there are values $v_1, \ldots, v_m$ and a rule $f\ \ell_1 \cdots \ell_n \to r$ with $n \leq m$ and substitution γ such that each $s_i \to^* v_i = \ell_i\gamma$ and $(r\gamma)\ v_{n+1} \cdots v_m \to^* w$

Note that rule selection is non-deterministic; a choice operator might for instance be implemented by having rules `choose` $x\ y \to x$ and `choose` $x\ y \to y$.

2.2 Cons-free Programs

Since the purpose of this research is to find groups of programs which can handle *restricted* classes of Turing-computable problems, we must impose certain limitations. In particular, we will limit interest to *cons-free* programs:

▶ **Definition 1.** A rule $\ell \to r$ is cons-free if for all subterms $r \trianglerighteq s$ of the form $s = c\ s_1 \cdots s_n$ with $c \in \mathcal{C}$, we have: $s \in \mathcal{DA}$ or $\ell \rhd s$. A program is cons-free if all its rules are.

This definition follows those for cons-free term rewriting in [3, 5] in generalising Jones' definition in [4]; the latter fixes the constructors in the program and therefore simply requires that the only non-constant constructor, `cons`, does not occur in any right-hand side.

In a cons-free program, if $v_1, \ldots, v_n, w$ are all data terms, then any data term occurring in the derivation of $f\ v_1 \cdots v_n \to^* w$ is a subterm of some v_i. This includes the result w.

3 Turing Machines and decision problems

In this paper, we particularly consider complexity classes of *decision problems*. A decision problem is a set $A \subseteq \{0,1\}^+$. A deterministic Turing Machine *decides* A in time $P(n)$ if every evaluation starting with a tape $\sqcup x_1 \dots x_n \sqcup\sqcup \dots$ completes in at most $P(n)$ steps, ending in the Accept state if $x_1 \dots x_n \in A$ and in the Reject state otherwise.

Let $\exp_2^0(m) = m$ and $\exp_2^{k+1}(m) = \exp_2^k(2^m) = 2^{\exp_2^k(m)}$. The class $\texttt{EXP}^k\texttt{TIME}$ consists of those decision problems which can be decided in $P(n) \leq \exp_2^k(a \cdot n^b)$ steps for some a, b.

▶ **Definition 2.** A program $(\mathcal{C}, \mathcal{D}, \mathcal{R})$ with constructors $\texttt{true}, \texttt{false} : \texttt{bool}$, $[] : \texttt{list}$ and $::$ (denoted infix) of type $\texttt{bool} \Rightarrow \texttt{list} \Rightarrow \texttt{list}$, and a defined symbol $\texttt{start} : \texttt{list} \Rightarrow \texttt{bool}$ *accepts* a decision problem A if for all $\vec{x} = x_1 \dots x_n \in \{0,1\}^+$: $\vec{x} \in A$ iff $\texttt{start}\ (\overline{x_1}:: \dots ::\overline{x_n}::[]) \to^* \texttt{true}$, where $\overline{x_i} = \texttt{true}$ if $x_i = 1$ and $\texttt{false}$ if $x_i = 0$. (Note that it is not required that *all* evaluations end in $\texttt{true}$, just that there is at least one—and none if $x \notin A$).

4 A lower bound for expressivity

To give a lower bound on expressivity, we consider the following result paraphrased from [4]:

▶ **Lemma 3.** *Suppose that, given an input list* $cs ::= \overline{x_1}:: \dots ::\overline{x_n}::[]$ *of length* n*, we have a representation of* $0, \dots, P(n)$*, symbols* $\texttt{seed}, \texttt{pred}, \texttt{zero} \in \mathcal{D}$*, and cons-free rules* $\mathcal{R}$ *with:*

- $\texttt{seed}\ cs \to^* v$ *for* v *a value representing* $P(n)$
- *if* v *represents* $i > 0$*, then* $\texttt{pred}\ cs\ v \to^* w$ *for* w *a value representing* $i - 1$
- *if* v *represents* i*, then* $\texttt{zero}\ cs\ i \to^* \texttt{true}$ *iff* $i = 0$ *and* $\texttt{zero}\ cs\ i \to^* \texttt{false}$ *iff* $i \neq 0$

Then any problem which can be decided in time $P(n)$ *is accepted by a cons-free program whose data order is the same as that of* $\mathcal{R}$*, and which is deterministic iff* $\mathcal{R}$ *is.*

Proof Idea. By simulating an evaluation of a Turing Machine. This simulation encodes all transitions of the machine as rules; a transition from state i to state j, reading symbol r, writing w and moving to the right is encoded by a rule $\texttt{transition i r} \to (\texttt{j}, (\texttt{w}, \texttt{R}))$. In addition, there are rules for $\texttt{state}\ cs\ n$—which returns the state the machine is in at time n—, $\texttt{position}\ cs\ n$—which returns the position of the tape reader—and $\texttt{tape}\ cs\ n\ p$—for the symbol on the tape at position p and time n. Rules are for instance:

$$\texttt{state}\ cs\ n \to \texttt{ifthenelse}\ (\texttt{zero}\ cs\ n)\ \texttt{Start}\ (\texttt{fst}\ (\texttt{transitionat}\ cs\ (\texttt{pred}\ cs\ n)))$$

This returns $\texttt{Start}$ at time 0, and otherwise the state reduced to in the last transition. ◀

▶ **Example 4.** For $P(n) = (n+1)^2 - 1$, we can represent $i \in \{0, \dots, P(n)\}$ as any pair (l_1, l_2) of lists, where $i = |l_1| \cdot (n+1) + |l_2|$. For the counting functions, we define:

$$\begin{array}{rcl@{\qquad}rcl} \texttt{seed}\ cs & \to & ([],[]) & \texttt{zero}\ cs\ ([],[]) & \to & \texttt{true} \\ \texttt{pred}\ cs\ (xs, y::ys) & \to & (xs, ys) & \texttt{zero}\ cs\ (xs, y::ys) & \to & \texttt{false} \\ \texttt{pred}\ cs\ (x::xs, []) & \to & (xs, cs) & \texttt{zero}\ cs\ (x::xs, []) & \to & \texttt{false} \end{array}$$

▶ **Lemma 5.** *For any* $a, b > 0, k \geq 0$*,there are cons-free, deterministic rules* $\mathcal{R}_{a,b}^k$ *defining counting functions as in Lemma 3 such that, for* $P(n) = \exp_2^k(a \cdot n^b) - 1$*, the numbers* $\{0, \dots, P(n)\}$ *can be represented. All function variables in* $\mathcal{R}_{a,b}^k$ *have a type* $\sigma \Rightarrow \texttt{bool}$*.*

Proof Idea. For $k = 0$, we can count to $a \cdot n^b - 1$ using an approach much like Example 4. Given $\mathcal{R}_{a,b}^k$, which represents numbers as a type σ, we can define $\mathcal{R}_{a,b}^{k+1}$ by representing a number i with bit vector $b_0 \dots b_M$ (with $M = \exp_2^k(a \cdot n^b)$) as the function in $\sigma \Rightarrow \texttt{bool}$ which maps a "number" j to $\texttt{true}$ if $b_i = 1$ and to $\texttt{false}$ otherwise. ◀

The observation that the functional variables take only one input argument will be used in Lemma 8 below. The counting techniques from Example 4 and Lemma 5 originate from Jones' work. However, in a non-deterministic system, we can do significantly more:

▶ **Lemma 6.** *Let $P_0(n) := n$, and for $k \geq 0$, $P_{k+1}(n) := 2^{P_k(n)} - 1$. Then for each k, we can represent all $i \in \{0, \ldots, P_k(n)\}$ as a term of type $\mathtt{bool}^k \Rightarrow \mathtt{list}$, and accompanying counting functions $\mathtt{seed}_k, \mathtt{pred}_k$ and $\mathtt{zero}_k$ can be defined.*

Proof. The base case ($k = 0$) is Example 4. For larger k, let $i \in \{0, \ldots, 2^{P_k(n)} - 1\}$ have bit vector $b_1 \ldots b_{P_k(n)}$; we say $s : \mathtt{bool}^k \Rightarrow \mathtt{list}$ represents i at level k if for all $1 \leq j \leq P_k(n)$: $b_j = 1$ iff $s\ \mathtt{true} \to^* v$ for some v which represents j at level $k-1$, and $b_j = 0$ iff $s\ \mathtt{false} \to^* v$ for such v. This relies on non-determinism: $s\ \mathtt{true}$ reduces to a representation of *every* j with $b_j = 1$. A representation O of 0 at level $k-1$ is used as a default, e.g. $s\ \mathtt{false} \to^* O$ even if each $b_j = 0$. The `zero` and `pred` rules rely on testing bit values, using:

$\mathtt{bitset}_k\ cs\ F\ j \to \mathtt{bshelp}_k\ cs\ F\ j\ (\mathtt{equal}_{k-1}\ cs\ (F\ \mathtt{true})\ j)\ (\mathtt{equal}_{k-1}\ cs\ (F\ \mathtt{false})\ j)$
$\mathtt{bshelp}_k\ cs\ F\ j\ \mathtt{true}\ b \to \mathtt{true}$ $\quad$ $\mathtt{bshelp}_k\ cs\ F\ j\ b\ \mathtt{true} \to \mathtt{false}$
$\mathtt{bshelp}_k\ cs\ F\ j\ \mathtt{false}\ \mathtt{false} \to \mathtt{bitset}_k\ cs\ F\ j.$

These rules are non-terminating, but if F represents a number at level k, and j at level $k-1$, then $\mathtt{bitset}_k\ cs\ F\ j$ reduces to exactly one value: `true` if $b_j = 1$, and `false` if $b_j = 0$. ◀

Thus, we can count up to arbitrarily high numbers; by Lemma 3, every decision problem in `ELEMENTARY` is accepted by a non-deterministic cons-free program of data order 1.

To obtain a more fine-grained characterisation which still admits non-deterministic choice, we will therefore consider a restriction of cons-free programming which avoids Lemma 6.

▶ **Definition 7.** A cons-free program *has unary variable* if all variables occurring in any rule in $\mathcal{R}$ have a type ι or $\sigma \Rightarrow \iota$, with $o(\iota) = 0$.

Intuitively, in a program with unary variables, functional variables cannot be *partially applied*; thus, such variables represent a function mapping to *data*, and not to some complex structure. Note that the input type σ of a unary variable $x : \sigma \Rightarrow \iota$ is allowed to be a product $\sigma_1 \times \cdots \times \sigma_n$. Lemma 6 relies on non-unary variables, but Lemma 5 does not. We obtain:

▶ **Lemma 8.** *Any problem in $\mathtt{EXP}^k\mathtt{TIME}$ is accepted by a (non-deterministic) extended cons-free program of data order k.*

5 An upper bound for expressivity

To see that extended cons-free programs *characterise* the `EXPTIME` hierarchy, it merely remains to be seen that every decision problem that is accepted by a call-by-value cons-free program with unary variables and of data order k, can be solved by a deterministic Turing Machine—or, equivalently, an algorithm in pseudo code—running in polynomial time.

▶ **Algorithm 9** (Finding the values for given input in a fixed extended cons-free program $\mathcal{R}$).
Input: a term $\mathtt{start}\ v_1 \cdots v_n : \iota$ with each v_i a data term and $o(\iota) = 0$.
Output: all data terms w such that $\mathtt{start}\ v_1 \cdots v_n \to^* w$.

Let $\mathcal{B} := \bigcup_{1 \leq i \leq m} \{w \in \mathcal{DA} \mid v_i \trianglerighteq w\} \cup \bigcup_{\ell \to r \in \mathcal{R}} \{w \in \mathcal{DA} \mid r \trianglerighteq w\}$.

For all types σ occurring as data in $\mathcal{R}$, generate $[\![\sigma]\!]$ and a relation $\sqsupseteq$, as follows:

- $[\![\kappa]\!] = \{s \in \mathcal{B} \mid s : \kappa\}$ if κ is a sort; for $A, B \in [\![\kappa]\!]$, let $A \sqsupseteq B$ if $A = B$
- $[\![\sigma \times \tau]\!] = \{(A, B) \mid A \in [\![\sigma]\!] \wedge B \in [\![\tau]\!]\}$; $(A_1, A_2) \sqsupseteq (B_1, B_2)$ if $A_1 \sqsupseteq B_1$ and $A_2 \sqsupseteq B_2$
- $[\![\sigma \Rightarrow \tau]\!] = \mathcal{P}(\{(A, B) \mid A \in [\![\sigma]\!] \wedge B \in [\![\tau]\!]\})$; for $A, B \in [\![\sigma \Rightarrow \tau]\!]$ let $A \sqsupseteq B$ if $A \supseteq B$

For all $f : \sigma_1 \Rightarrow \ldots \Rightarrow \sigma_m \Rightarrow \iota \in \mathcal{D}$, note that we can safely assume that $\mathit{arity}(f) \geq m - 1$. For all such f, and all $A_1 \in [\![\sigma_1]\!], \ldots, A_m \in [\![\sigma_m]\!], v \in [\![\iota]\!]$, note down a statement: $f\ A_1 \cdots A_m \approx v$. If $\mathit{arity}(f) = m-1$, also note down $f\ A_1 \cdots A_{m-1} \approx O$ for all $O \in [\![\sigma_m \Rightarrow \iota]\!]$.

For all rules $\ell \to r$, all $s : \tau$ with $r \trianglerighteq s$ or $s = r\ x$, all $O \in [\![\tau]\!]$ and all substitutions γ mapping the variables $x : \sigma \in \mathit{Var}(s)$ to elements of $[\![\sigma]\!]$, note down a statement $s\gamma \approx O$. Mark all statements $x\gamma \approx O$ such that $x\gamma \sqsupseteq O$ as *confirmed*, and all other statements *unconfirmed*. Repeat the following steps until no new statements are confirmed anymore.

- For every unconfirmed statement $f\ A_1 \cdots A_n \approx O$, determine all rules $f\ \ell_1 \cdots \ell_k \to r$ (with $k = n$ or $k = n - 1$) and substitutions γ mapping $x : \sigma \in \mathit{Var}(f\ \ell_1 \cdots \ell_k)$ to an element of $[\![\sigma]\!]$, such that each $A_i = \ell_i\gamma$, and mark the statement as confirmed if $(r\ x_{k+1} \cdots x_n)\gamma[x_{k+1} := A_{k+1}, \ldots, x_n := A_n] \approx O$ is confirmed.
- For every unconfirmed statement $(F\ s)\gamma \approx O$, mark the statement as confirmed if there exists A with $(A, O) \in \gamma(F)$ and $s\gamma \approx A$ is confirmed.
- For every unconfirmed statement $(f\ s_1 \cdots s_n)\gamma \approx O$, mark it as confirmed if there are $A_1, \ldots, A_n$ such that both $f\ A_1 \cdots A_n \approx O$ and each $s_i\gamma \approx A_i$ are confirmed.

Then return all w such that $\mathtt{start}\ v_1 \cdots v_n \approx w$ is marked confirmed.

▶ **Lemma 10.** *Algorithm 9 is in* $\mathtt{EXP}^k\mathtt{TIME}$*—where k is the data order of $\mathcal{R}$—and returns the claimed output.*

Proof Idea. The complexity of Algorithm 9 is determined by the size of each $[\![\sigma]\!]$. The proof of soundness and completeness of the algorithm is more intricate; this fundamentally relies on replacing the values $f\ v_1 \cdots v_n$ with $n < \mathit{arity}(f)$ by subsets of the set of all tuples (A, w) with the property that, intuitively, $f\ v_1 \cdots v_n\ A \to^* w$. ◀

6 Conclusion

Thus, we obtain the following variation of Jones' result:

▶ **Theorem 11.** *A decision problem A is in* $\mathtt{EXP}^k\mathtt{TIME}$ *if and only if there is a cons-free program $\mathcal{R}$ of data order k and with unary variables, which accepts A. This statement holds whether or not the program is allowed to use non-deterministic choice.*

In addition, we have adapted Jones' language to be more permissive, admitting additional constructors and pattern matching. This makes it easier to specify suitable programs.

Using non-deterministic programs is a step towards further characterisations; in particular, we intend to characterise $\mathtt{NEXP}^k\mathtt{TIME} \subseteq \mathtt{EXP}^{k+1}\mathtt{TIME}$ using restricted non-deterministic cons-free programs of data order $k + 1$.

References

1 G. Bonfante. Some programming languages for logspace and ptime. In *AMAST '06*, volume 4019 of *LNCS*, pages 66–80, 2006.

2 S.A. Cook. Characterizations of pushdown machines in terms of time-bounded computers. *ACM*, 18(1):4–18, 1971.

3 D. de Carvalho and J. Simonsen. An implicit characterization of the polynomial-time decidable sets by cons-free rewriting. In *RTA-TLCA '14*, volume 8560 of *LNCS*, pages 179–193, 2014.

4 N. Jones. Life without cons. *JFP*, 11(1):5–94, 2001.

5 C. Kop and J. Simonsen. Complexity hierarchies and higher-order cons-free rewriting. In *FSCD '16*, volume 52 of *LIPIcs*, pages 23:1–23:18, 2016.

The Generalized Subterm Criterion in TTT2*

Christian Sternagel

University of Innsbruck, Austria
christian.sternagel@uibk.ac.at

Abstract

We present an SMT encoding of a generalized version of the subterm criterion and evaluate its implementation in TTT2.

1998 ACM Subject Classification F.4.2 Grammars and Other Rewriting Systems

Keywords and phrases termination, subterm criterion, SMT encodings

1 Preliminaries

We assume basic familiarity with term rewriting [1] in general and the dependency pair framework [3] for proving termination in particular. We start with a recap of terminology and notation that we use in the remainder.

By $\mathcal{M}(A)$, we denote the set of *finite multisets* ranging over elements from the set A. We write $M(x)$ for the *multiplicity* (i.e., number of occurrences) of x in the multiset M, use $+$ for multiset sum, but otherwise use standard set-notation.

Given a relation $\succ$, its *restriction to the set* A, written $\succ_{\downarrow A}$, is the relation defined by the set $\{(x, y) \mid x \succ y, x \in A, y \in A\}$. Moreover, for any function f, we use $x \succ^f y$ as a shorthand for $f(x) \succ f(x)$.

The *multiset extension* $\succ_{\mathsf{mul}}$ of a given relation $\succ$ is defined by:

$$M \succ_{\mathsf{mul}} N \text{ iff } \exists X\ Y\ Z.\ X \neq \varnothing, M = X + Z, N = Y + Z, \forall y \in Y.\ \exists x \in X.\ x \succ y$$

A useful fact about the multiset extension is that we may always "maximize" the common part Z in the above definition.

▶ **Lemma 1.** *Consider an irreflexive and transitive relation $\succ$ and multisets M, N such that $M \succ_{\mathsf{mul}} N$. Moreover, let $X = M - M \cap N$ and $Y = N - M \cap N$. Then $X \neq \varnothing$ and $\forall y \in Y.\ \exists x \in X.\ x \succ y$.*

While intuitively obvious, a rigorous proof of this fact does not seem to be widely known.[1] In preparation for the proof, we recall the following easy fact about finite relations.

▶ **Lemma 2.** *Every finite, irreflexive, and transitive relation is well-founded.*

Proof. Let $\succ$ be a finite, irreflexive, and transitive relation. For the sake of a contradiction, assume that $\succ$ is not well-founded. Then there is an infinite sequence $a_1 \succ a_2 \succ a_3 \succ \cdots$ whose elements are in the finite (since $\succ$ is finite) field of $\succ$. But then, by the (infinite) pigeonhole principle, there is some recurring element a_i, i.e., $\cdots \succ a_i \succ \cdots \succ a_i \succ \cdots$. By transitivity we obtain $a_i \succ a_i$ contradicting the irreflexivity of $\succ$. ◀

* This work was supported by FWF (Austrian Science Fund) project P27502.

[1] An alternative proof of this fact is indicated in Vincent van Oostrom's PhD thesis [7].

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 11; pp. 11:1–11:5.

Noting that the converse of any finite, irreflexive, and transitive relation is again finite, irreflexive, and transitive, Lemma 2 allows us to employ well-founded induction where the induction hypothesis holds for "bigger" elements, as exemplified in the following proof.

Proof of Lemma 1. Since $M \succ_{\mathsf{mul}} N$ we obtain $I \neq \varnothing$, J, and K such that $M = I + K$, $N = J + K$, and $\forall j \in J.\, \exists i \in I.\, i \succ j$. Let $A = I - I \cap J$, $B = J - I \cap J$, and consider the finite set D of elements occurring in either of I and J. Now, appealing to Lemma 2, we employ well-founded induction with respect to $\prec_{\downarrow D}$ in order to prove:

$$\forall j \in J.\, \exists a \in A.\, a \succ j \tag{$\dagger$}$$

Thus we assume $j \in J$ for some arbitrary but fixed j and obtain the induction hypothesis (IH) $\forall c \succ_{\downarrow D} j.\, c \in J \longrightarrow \exists a \in A.\, a \succ c$. From $j \in J$ we obtain an $i \in I$ with $i \succ j$. Now if $i \in A$, then we are done. Otherwise, $i \in J$ and by IH we obtain an $a \in A$ with $a \succ i$. Since $\succ$ is transitive, this implies $a \succ j$, concluding the proof of ($\dagger$). But then also $\forall b \in B.\, \exists a \in A.\, x \succ b$ and $A \neq \varnothing$. We conclude by noting the following two equalities:

$$X = M - M \cap N = (I + K) - (I + K) \cap (J + K) = I - I \cap J = A,$$
$$Y = N - M \cap N = (J + K) - (I + K) \cap (J + K) = J - I \cap J = B. \qquad \blacktriangleleft$$

2 A Generalized Subterm Criterion

Recall the subterm criterion – originally by Hirokawa and Middeldorp [4] and later reformulated as a processor for the dependency pair framework – which is a particularly elegant technique (due to its simplicity and the fact that the $\mathcal{R}$-component of a dependency pair problem $(\mathcal{P}, \mathcal{R})$ may be ignored).

▶ **Definition 3** (Simple projections). A *simple projection* is a function $\pi : \mathcal{F} \to \mathbb{N}$ that maps every n-ary function symbol f to some natural number $\pi(f) \in \{1, \dots, n\}$. Applying a simple projection to a term is defined by $\pi(f(t_1, \dots, t_n)) = t_{\pi(f)}$.

▶ **Theorem 4.** *If $\mathcal{P} \subseteq \trianglerighteq^{\pi}$ for simple projection π, then $(\mathcal{P}, \mathcal{R})$ is finite iff $(\mathcal{P} \setminus \triangleright^{\pi}, \mathcal{R})$ is.* ◀

Recall that the appropriate notion of *finiteness* for the subterm criterion is "the absence of minimal infinite chains."

For an AC-variant of the subterm criterion (i.e., a variant for rewriting modulo associative and/or commutative function symbols), Yamada et al. [8] generalized simple projections to so-called multiprojections.

▶ **Definition 5** (Multiprojections). A *multiprojection* is a function $\pi : \mathcal{F} \to \mathcal{M}(\mathbb{N})$ that maps every n-ary function symbol f to a multiset $\pi(f) \subseteq \mathcal{M}(\{1, \dots, n\})$. Applying a multiprojection to a term yields a multiset of terms as follows:

$$\pi(t) = \begin{cases} \pi(t_{i_1}) + \cdots + \pi(t_{i_k}) & \text{if } t = f(t_1, \dots, t_n) \text{ and } \pi(f) = \{i_1, \dots, i_k\} \neq \varnothing, \\ \{t\} & \text{otherwise.} \end{cases}$$

We write $s \trianglerighteq^{\pi}_{\mathsf{mul}} t$ if either $s \triangleright^{\pi}_{\mathsf{mul}} t$ or $\pi(s) = \pi(t)$.

A compromise between simple projections and full multiprojections is to allow recursive projections (possibly through defined symbols). While theoretically subsumed by multiprojections, we included such recursive projections in our experiments in order to assess their performance in practice.

The following is a specialization of the AC subterm criterion by Yamada et al. [8, Theorem 33] to the non-AC case.

▶ **Theorem 6.** *Let π be a multiprojection such that $\mathcal{P} \subseteq \trianglerighteq^{\pi}_{\mathsf{mul}}$ and $f(\ldots) \trianglerighteq^{\pi}_{\mathsf{mul}} r$ for all $f(\ldots) \to r \in \mathcal{R}$ with $\pi(f) \neq \varnothing$. Then $(\mathcal{P}, \mathcal{R})$ is finite iff $(\mathcal{P} \setminus \rhd^{\pi}_{\mathsf{mul}}, \mathcal{R})$ is.* ◀

This result (which is also formalized in IsaFoR [6]) states the soundness of a generalized version of the subterm criterion and thus gives the theoretical backing for implementing such a technique in a termination tool. In the following we are concerned with the more practical problem of an efficient implementation.

That is, given a DP problem $(\mathcal{P}, \mathcal{R})$ we want to find a multiprojection π that satisfies the conditions of Theorem 6 and orients at least one rule of $\mathcal{P}$ strictly by $\rhd^{\pi}_{\mathsf{mul}}$.

Since the problem of finding such a multiprojection seems similar to the problem of finding an appropriate argument filter for a reduction pair [2], and the latter has been successfully tackled by various kinds of SAT and SMT encodings, we take a similar approach.

3 Implementation and Experiments

There are basically two issues that have to be considered: (1) how to encode a multiprojection π and thereby the multiset $\pi(s)$, and (2) how to encode the comparison between two encodings of multisets with respect to the multiset extension of $\rhd$.

In the following we use lowercase sans serif for propositional and arithmetical variables, and UPPERCASE SANS SERIF for functions that result in formulas.

Encoding Multiprojections. We encode the multiplicity of a term t in the multiset $\pi(s)$, which is 0 if t does not occur in $\pi(s)$ at all, by $\mathsf{M}_s(t) = \mathsf{MUL}(1, s, t)$. The latter is defined as follows

$$\mathsf{MUL}(w, s, t) = \begin{cases} \left(\bigwedge_{1 \le i \le n} \neg \mathsf{p}^i_f\right) \mathbin{?} w : 0 & \text{if } s = t = f(t_1, \ldots, t_n) \\ w & \text{if } s = t \text{ and } t \text{ is a variable} \\ \sum_{1 \le i \le n} (\mathsf{p}^i_f \mathbin{?} \mathsf{MUL}(w \cdot \mathsf{w}^i_f, s_i, t) : 0) & \text{if } t \lhd s = f(s_1, \ldots, s_n) \\ 0 & \text{otherwise} \end{cases}$$

where $b \mathbin{?} t : e$ denotes *if b then t else e* and the intended meaning of variables is that $\mathsf{p}^i_f = \top$ precisely when π *projects to the i-th argument of f*, in which case w^i_f gives the *weight of i in* $\pi(f)$, i.e., its number of occurrences in $\pi(f)$.[2]

Encoding Multiset Comparison. Now consider the problem of finding π such that $s \rhd^{\pi}_{\mathsf{mul}} t$ for given terms s and t. Noting that, independent of the exact π, $\pi(s)$ and $\pi(t)$ are multisets over the finite set of subterms of s and t, it suffices to find an encoding for comparing multisets over finite domains. This allows us to make use of the following observation.

▶ **Lemma 7** (Comparing multisets over finite domains). *Let D be a finite set, and $M, N \subseteq \mathcal{M}(D)$. Then, for irreflexive and transitive $\succ$, $M \succ_{\mathsf{mul}} N$ is equivalent to*

$$\forall d \in D.\, \mathrm{upper}(d) \longrightarrow M(d) \geq N(d) \text{ and } M \neq N \qquad (\star)$$

where $\mathrm{upper}(x)$ *iff* $\forall d \in D.\, d \succ x \longrightarrow M(d) = N(d)$.

[2] In experiments, replacing $\mathsf{p}^i_f = \top$ by $\mathsf{w}^i_f > 0$ resulted in a slightly increased number of timeouts.

Proof. We start with the direction from ($\star$) to $M \succ_{\mathsf{mul}} N$. Assume ($\star$) for M and N, and define the multisets $Z = \{x \in M \cap N \mid \mathrm{upper}(x)\}$, $X = M - Z$, and $Y = N - Z$ (i.e., $M = X + Z$ and $N = Y + Z$). Then, appealing to Lemma 2, we use well-founded induction with respect to $\prec_{\downarrow D}$ in order to prove

$$\forall y \in Y.\, \exists x \in X.\, x \succ y \tag{‡}$$

Thus we assume $y \in Y$ for some arbitrary but fixed y and obtain the induction hypothesis (IH) $\forall z \succ_{\downarrow D} y.\, z \in Y \longrightarrow \exists x \in X.\, x \succ z$. Also note that $\neg\mathrm{upper}(y)$, since otherwise $M(y) \geq N(y)$ by ($\star$) and thus $Z(y) = N(y)$, contradicting $y \in Y$. Therefore, we obtain $z \succ y$ with $M(z) \neq N(z)$ by definition of upper. Now, either $M(z) > N(z)$ or $N(z) > M(z)$. In the former case $z \in X$ and we are done. In the latter case $z \in Y$ and thus we obtain an $x \in X$ such that $x \succ z$ by IH and conclude (‡) by transitivity of $\succ$. It remains to show $X \neq \varnothing$. Since $M \neq N$ there is some x with $M(x) \neq N(x)$. If $M(x) > N(x)$, then $x \in X$ and we are done. Otherwise, $N(x) > M(x)$ and thus $x \in Y$ and we conclude by invoking (‡).

For the other direction, assume $M \succ_{\mathsf{mul}} N$. Then for $Z = M \cap N$, $X = M - Z$, and $Y = M - Z$, we have $X \neq \varnothing$, $X \cap Y = \varnothing$, $M = X + Z$, $N = Y + Z$ and $\forall y \in Y. \exists x \in X. x \succ y$, using Lemma 1. This further implies $M \neq N$. Now assume $d \in D$ and $\mathrm{upper}(d)$. Then either $d \in Y$ or $d \notin Y$. In the latter case, clearly $M(d) \geq N(d)$, and we are done. In the former case, we obtain an $x \in X$ with $x \succ d$. Moreover, since $X \cap Y = \varnothing$, we have $x \notin Y$. But then $M(x) \neq N(x)$, contradicting $\mathrm{upper}(d)$. ◀

Encoding the Generalized Subterm Criterion. Putting everything together we obtain the encoding

$$(\forall s \to t \in \mathcal{P}.\, \mathsf{GEQ}(s,t)) \wedge (\exists s \to t \in \mathcal{P}.\, \mathsf{NEQ}(s,t)) \wedge$$
$$(\forall s \to t \in \mathcal{R}.\, \mathsf{RT}(s) \longrightarrow \mathsf{GEQ}(s,t)) \wedge (\forall f \in \mathcal{F}(\mathcal{P},\mathcal{R}).\, \mathsf{SAN}(f))$$

where

$$\begin{aligned}
\mathsf{GEQ}(s,t) &\text{ iff } \forall u \in \mathcal{S}\mathsf{ub}(s,t).\, \mathsf{UPPER}(u) \longrightarrow \mathsf{M}_s(u) \geq \mathsf{M}_t(u)\\
\mathsf{UPPER}(u) &\text{ iff } \forall v \in \mathcal{S}\mathsf{ub}(s,t).\, v \rhd u \longrightarrow \mathsf{M}_s(v) = \mathsf{M}_t(v)\\
\mathsf{NEQ}(s,t) &\text{ iff } \neg(\forall u \in \mathcal{S}\mathsf{ub}(s,t).\, \mathsf{M}_s(u) = \mathsf{M}_t(u))\\
\mathsf{RT}(f(s_1,\ldots,s_n)) &\text{ iff } \exists 1 \leq i \leq n.\, \mathsf{p}^i_f.\\
\mathsf{SAN}(f) &\text{ iff } \bigwedge_{1 \leq i \leq \mathsf{arity}(f)} \left(\mathsf{p}^i_f \longrightarrow \mathsf{w}^i_f > 0\right)
\end{aligned}$$

Here $\mathcal{S}\mathsf{ub}(s,t)$ denotes the set of all (i.e., including s and t themselves) subterms of s and t, and SAN is a "sanity check" that makes sure that propositional and arithmetical variables play well together. Every satisfying assignment gives rise to a multiprojection π satisfying the conditions of Theorem 6.

Experiments. We conducted experiments in order to assess our implementation. To this end we took all the 1498 TRSs in the standard (as in "standard term rewriting") category of the termination problem database (TPDB) version 10.3 and tried to prove their termination with the following strategy: first compute dependency pairs, then compute the estimated dependency graph $\mathcal{G}$, and finally try repeatedly to either decompose $\mathcal{G}$ into strongly connected components or apply the subterm criterion. For the subterm criterion we tried either *simple projections* (simple), *recursive projections* (recursive), *multiprojection* (multi), or a parallel combination of those (all).

Table 1 Experiments on 1498 standard TRSs of TPDB 10.3

	Yes		Maybe		Timeout		
Projections	#	(sec)	#	(sec)	#	(sec)	Total (sec)
simple	265	31.1	1184	226.8	49	254.0	502.9
recursive	292	35.4	1155	240.4	51	255.0	530.9
multi	351	61.2	1081	419.0	66	330.0	810.2
all	352	30.4	1099	230.3	47	235.0	495.7

In summary, the parallel combination of different kinds of projections results in a significant increase of power (i.e., number of yeses) and does not have a negative impact on the speed, compared to the original implementation of the subterm criterion (simple) of TTT2 [5].

Encouraged by this results, we incorporated our new implementation also into the competition strategy of TTT2 and compared it to its 2015 competition version. In this way, we were able to obtain 12 additional yeses. However, each of those 12 systems could already be handled by some other termination tool in the 2015 termination competition.

Acknowledgments. We thank Vincent van Oostrom for pointing us to Lemma 7 and Bertram Felgenhauer for helpful discussion concerning MUL. We further thank the Austrian Science Fund (FWF project P27502) for supporting this work.

References

1 Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998. 10.1017/CBO9781139172752.

2 Michael Codish, Peter Schneider-Kamp, Vitaly Lagoon, René Thiemann, and Jürgen Giesl. SAT solving for argument filterings. In *Proc. 13th LPAR*, volume 4246 of *LNCS*, pages 30–44. Springer, 2006. 10.1007/11916277_3.

3 Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. 11th LPAR*, volume 3452 of *LNCS*, pages 301–331. Springer, 2005. 10.1007/978-3-540-32275-7_21.

4 Nao Hirokawa and Aart Middeldorp. Dependency pairs revisited. In *Proc. 15th RTA*, volume 3091 of *LNCS*, pages 249–268. Springer, 2004. 10.1007/978-3-540-25979-4_18.

5 Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean Termination Tool 2. In *Proc. 20th RTA*, volume 5595 of *LNCS*, pages 295–304. Springer, 2009. 10.1007/978-3-642-02348-4_21.

6 René Thiemann and Christian Sternagel. Certification of termination proofs using CeTA. In *Proc. 22nd TPHOLs*, volume 5674 of *LNCS*, pages 452–468. Springer, 2009. 10.1007/978-3-642-03359-9_31.

7 Vincent van Oostrom. *Confluence for Abstract and Higher-Order Rewriting.* PhD thesis, Vrije Universiteit, Amsterdam, 1994.

8 Akihisa Yamada, Christian Sternagel, René Thiemann, and Keiichirou Kusakari. AC dependency pairs revisited. In *Proc. 25th CSL*, LIPIcs. Schloss Dagstuhl, 2016. to appear.

A Characterization of Quasi-Decreasingness*

Thomas Sternagel[1] and Christian Sternagel[1]

1 **University of Innsbruck, Austria**
{thomas,christian}.sternagel@uibk.ac.at

1 Introduction

In 2010 Schernhammer and Gramlich [10] showed that quasi-decreasingness of a DCTRS $\mathcal{R}$ is equivalent to μ-termination of its context-sensitive unraveling $U_{\mathsf{CS}}(\mathcal{R})$ on original terms. While the direction that quasi-decreasingness of $\mathcal{R}$ implies μ-termination of $U_{\mathsf{CS}}(\mathcal{R})$ on original terms is shown directly; the converse – facilitating the use of context-sensitive termination tools like MU-TERM [1] and VMTL [9] – employs the additional notion of context-sensitive quasi-reductivity of $\mathcal{R}$. In the following, we give a direct proof of the fact that μ-termination of $U_{\mathsf{CS}}(\mathcal{R})$ on original terms implies quasi-decreasingness of $\mathcal{R}$. Moreover, we report our experimental findings on DCTRSs from the confluence problems database (Cops),[1] extending the experiments of Schernhammer and Gramlich.

Contribution. A direct proof that μ-termination of a CSRS $U_{\mathsf{CS}}(\mathcal{R})$ on original terms implies quasi-decreasingness of the DCTRS $\mathcal{R}$. New experiments on a recent DCTRS collection.

2 Preliminaries

We assume familiarity with the basic notions of (conditional and context-sensitive) term rewriting [3, 6, 8], but shortly recapitulate terminology and notation that we use in the remainder. Given two arbitrary binary relations $\to_\alpha$ and $\to_\beta$, we write ${}_\alpha\!\leftarrow$, $\to_\alpha^+$, $\to_\alpha^*$ for the *inverse*, the *transitive closure*, and the *reflexive transitive closure* of $\to_\alpha$, respectively. The relation obtained by considering $\to_\alpha$ *relative to* $\to_\beta$, written $\to_{\alpha/\beta}$, is defined by $\to_\beta^* \cdot \to_\alpha \cdot \to_\beta^*$. We use $\mathcal{V}(\cdot)$ to denote the set of variables occurring in a given syntactic object, like a term, a pair of terms, a list of terms, etc. The set of terms $\mathcal{T}(\mathcal{F},\mathcal{V})$ over a given signature of function symbols $\mathcal{F}$ and set of variables $\mathcal{V}$ is defined inductively: $x \in \mathcal{T}(\mathcal{F},\mathcal{V})$ for all variables $x \in \mathcal{V}$, and for every n-ary function symbol $f \in \mathcal{F}$ and terms $t_1,\ldots,t_n \in \mathcal{T}(\mathcal{F},\mathcal{V})$ also $f(t_1,\ldots,t_n) \in \mathcal{T}(\mathcal{F},\mathcal{V})$. A *deterministic oriented 3-CTRS (DCTRS)* $\mathcal{R}$ is a set of conditional rewrite rules of the shape $\ell \to r \Leftarrow c$ where ℓ and r are terms and c is a possibly empty sequence of pairs of terms $s_1 \approx t_1,\ldots,s_n \approx t_n$. For all rules in $\mathcal{R}$ we have that $\ell \notin \mathcal{V}$, $\mathcal{V}(r) \subseteq \mathcal{V}(\ell,c)$, and $\mathcal{V}(s_i) \subseteq \mathcal{V}(\ell,t_1,\ldots,t_{i-1})$ for all $1 \leqslant i \leqslant n$. The rewrite relation induced by a DCTRS $\mathcal{R}$ is structured into levels. For each level i, a TRS $\mathcal{R}_i$ is defined recursively by $\mathcal{R}_0 = \varnothing$ and $\mathcal{R}_{i+1} = \{\ell\sigma \approx r\sigma \mid \ell \to r \Leftarrow c \in \mathcal{R} \wedge \forall s \approx t \in c.\ s\sigma \to_{\mathcal{R}_i}^* t\sigma\}$ where for a given TRS $\mathcal{S}$, $\to_{\mathcal{S}}$ denotes the induced rewrite relation (i.e., its closure under contexts and substitutions). Then the rewrite relation of $\mathcal{R}$ is $\to_{\mathcal{R}} = \bigcup_{i\geqslant 0} \to_{\mathcal{R}_i}$. We have $\mathcal{R} = \mathcal{R}_{\mathsf{c}} \uplus \mathcal{R}_{\mathsf{u}}$ where $\mathcal{R}_{\mathsf{c}}$ denotes the subset of rules with non-empty conditional part ($n > 0$) and $\mathcal{R}_{\mathsf{u}}$ the subset of unconditional rules ($n = 0$). A DCTRS $\mathcal{R}$ over signature $\mathcal{F}$ is *quasi-decreasing* if there is a well-founded order $\succ$ on $\mathcal{T}(\mathcal{F},\mathcal{V})$ such that $\succ = (\succ \cup \rhd)^+$, $\to_{\mathcal{R}} \subseteq \succ$, and for all rules $\ell \to r \Leftarrow s_1 \approx t_1,\ldots,s_n \approx t_n$ in $\mathcal{R}$, all substitutions $\sigma\colon \mathcal{V} \to \mathcal{T}(\mathcal{F},\mathcal{V})$, and $0 \leqslant i < n$, if $s_j\sigma \to_{\mathcal{R}}^* t_j\sigma$ for all $1 \leqslant j \leqslant i$ then $\ell\sigma \succ s_{i+1}\sigma$.

* The research described in this paper is supported by FWF (Austrian Science Fund) project P27502.
[1] http://cops.uibk.ac.at

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 12; pp. 12:1–12:5.

Given a DCTRS $\mathcal{R}$ its *unraveling* $U(\mathcal{R})$ (cf. [8, p. 212]) is defined as follows. For each conditional rule $\rho\colon \ell \to r \Leftarrow s_1 \approx t_1, \ldots, s_n \approx t_n$ (where $n > 0$) we introduce n fresh function symbols $U_1^\rho, \ldots, U_n^\rho$ and generate the set of $n+1$ unconditional rules $U(\rho)$ as follows

$$\begin{aligned} \ell &\to U_1^\rho(s_1, \mathsf{v}(\ell)) \\ U_1^\rho(t_1, \mathsf{v}(\ell)) &\to U_2^\rho(s_2, \mathsf{v}(\ell), \mathsf{ev}(t_1)) \\ &\vdots \\ U_n^\rho(t_n, \mathsf{v}(\ell), \mathsf{ev}(t_1, \ldots, t_{n-1})) &\to r \end{aligned}$$

where v and ev denote functions that yield the respective sequences of elements of $\mathcal{V}$ and $\mathcal{EV}$ in some arbitrary but fixed order, and $\mathcal{EV}(t_i) = \mathcal{V}(t_i) \setminus \mathcal{V}(\ell, t_1, \ldots, t_{i-1})$ denotes the *extra variables* of the right-hand side of the ith condition. Finally the unraveling of the DCTRS is $U(\mathcal{R}) = \mathcal{R}_{\mathsf{u}} \cup \bigcup_{\rho \in \mathcal{R}_{\mathsf{c}}} U(\rho)$.

A *context-sensitive rewrite system* (CSRS) is a TRS (over signature $\mathcal{F}$) together with a replacement map $\mu\colon \mathcal{F} \to 2^{\mathbb{N}}$ that restricts the argument positions of each function symbol in $\mathcal{F}$ at which we are allowed to rewrite. A position p is *active* in a term t if either $p = \epsilon$, or $p = iq$, $t = f(t_1, \ldots, t_n)$, $i \in \mu(f)$, and q is active in t_i. The set of active positions in a term t is denoted by $\mathcal{P}\mathsf{os}_\mu(t)$. Given a CSRS $\mathcal{R}$ a term s μ-rewrites to a term t, written $s \to_\mu t$, if $s \to_{\mathcal{R}} t$ at some position p and $p \in \mathcal{P}\mathsf{os}_\mu(s)$. A CSRS is called *$\mu$-terminating* if its context-sensitive rewrite relation is terminating. The (proper) subterm relation with respect to replacement map μ, written $\rhd_\mu$, restricts the ordinary subterm relation to active positions.

We conclude this section by recalling the notion of context-sensitive quasi-reductivity in an attempt to further appreciation for a proof without this notion.

▶ **Definition 1.** A CSRS $\mathcal{R}$ over signature $\mathcal{F}$ is *context-sensitively quasi-reductive* if there is an extended signature $\mathcal{F}' \supseteq \mathcal{F}$, a replacement map μ (with $\mu(f) = \{1, \ldots, n\}$ for every n-ary $f \in \mathcal{F}$), and a μ-monotonic, well-founded partial order $\succ_\mu$ on $\mathcal{T}(\mathcal{F}', \mathcal{V})$ such that for every rule $\ell \to r \Leftarrow s_1 \approx t_1, \ldots, s_k \approx t_k$, every substitution $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$, and every $0 \leqslant i \leqslant k-1$:

- $\ell\sigma \; (\succ_\mu \cup \rhd_\mu)^+ \; s_{i+1}\sigma$ whenever $s_j\sigma \succeq_\mu t_j\sigma$ for every $1 \leqslant j \leqslant i$, and
- $\ell\sigma \succ_\mu r\sigma$ whenever $s_j\sigma \succeq_\mu t_j\sigma$ for every $1 \leqslant j \leqslant k$.

3 Characterization

In order to present our main result (the proof of Theorem 5 below) we first restate some definitions and theorems which we will use in the proof.

The usual unraveling is extended by a replacement map in order to restrict reductions in U-symbols to the first argument position [10, Definition 4].

▶ **Definition 2** (Unraveling $U_{\mathsf{CS}}(\mathcal{R})$)**.** The *context-sensitive unraveling* $U_{\mathsf{CS}}(\mathcal{R})$ is the unraveling $U(\mathcal{R})$ together with the replacement map μ such that $\mu(f) = \{1, \ldots, k\}$ if $f \in \mathcal{F}$ with arity k and $\mu(f) = \{1\}$ otherwise. We say that the resulting CSRS is *μ-terminating on original terms* [10, Definition 7], if there is no infinite $U_{\mathsf{CS}}(\mathcal{R})$-reduction starting from a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$.

Simulation completeness of $U_{\mathsf{CS}}(\mathcal{R})$ (i.e., that every $\mathcal{R}$-step can be simulated by a $U_{\mathsf{CS}}(\mathcal{R})$-reduction) can be shown by induction on the level of a conditional rewrite step [10, Theorem 1].

▶ **Theorem 3** (Simulation completeness)**.** *For a DCTRS $\mathcal{R}$ we have $\to_{\mathcal{R}} \subseteq \to^+_{U_{\mathsf{CS}}(\mathcal{R})}$.* ◀

Furthermore, we need the following auxiliary result.

▶ **Lemma 4.** *For any context-sensitive rewrite relation $\to_\mu$ induced by the replacement map μ, $\rhd_\mu$ commutes over $\to_\mu$, i.e., $\rhd_\mu \cdot \to_\mu \subseteq \to_\mu \cdot \rhd_\mu$.*

Proof. Assume $s \rhd_\mu t \to_\mu u$ for some terms s, t, and u. Then $s = C[t] \rhd_\mu t \to_\mu u$ for some nonempty context C. Thus we conclude by $C[t] \to_\mu C[u] \rhd_\mu u$. ◀

With this we are finally able to prove our main result.

▶ **Theorem 5.** *If the CSRS $U_{\mathsf{CS}}(\mathcal{R})$ is μ-terminating on original terms then the DCTRS $\mathcal{R}$ is quasi-decreasing.*

Proof. Assume that $U_{\mathsf{CS}}(\mathcal{R})$ is μ-terminating on original terms. We define an order $\succ$ on $\mathcal{T}(\mathcal{F}, \mathcal{V})$

$$\succ \stackrel{\text{def}}{=} (\to_{U_{\mathsf{CS}}(\mathcal{R})} \cup \rhd_\mu)^+ \cap (\mathcal{T}(\mathcal{F}, \mathcal{V}) \times \mathcal{T}(\mathcal{F}, \mathcal{V})) \tag{$\star$}$$

and show that it satisfies the four properties from the definition of quasi-decreasingness:

1. We start by showing that $\succ$ is well-founded on $\mathcal{T}(\mathcal{F}, \mathcal{V})$. Assume, to the contrary, that $\succ$ is not well-founded. Then we have an infinite sequence

   $$t_1 \succ t_2 \succ t_3 \succ \dots \tag{$\dagger$}$$

   where all $t_i \in \mathcal{T}(\mathcal{F}, \mathcal{V})$. By definition $\rhd_\mu$ is well-founded. Moreover, since $U_{\mathsf{CS}}(\mathcal{R})$ is μ-terminating on original terms, $\to_{U_{\mathsf{CS}}(\mathcal{R})}$ is well-founded on $\mathcal{T}(\mathcal{F}, \mathcal{V})$. Further note that every $\to_{U_{\mathsf{CS}}(\mathcal{R})}$-terminating element (hence every term in $\mathcal{T}(\mathcal{F}, \mathcal{V})$) is $\to_{U_{\mathsf{CS}}(\mathcal{R})/\rhd_\mu}$-terminating, since by a repeated application of Lemma 4 every infinite reduction $t_1 \to_{U_{\mathsf{CS}}(\mathcal{R})/\rhd_\mu} t_2 \to_{U_{\mathsf{CS}}(\mathcal{R})/\rhd_\mu} \cdots$ starting from a term $t_1 \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ can be transformed into an infinite $\to_{U_{\mathsf{CS}}(\mathcal{R})}$-reduction, contradicting well-foundedness of $\to_{U_{\mathsf{CS}}(\mathcal{R})}$ on $\mathcal{T}(\mathcal{F}, \mathcal{V})$. We conclude by analyzing the following two cases:
   - Either ($\dagger$) contains $\to_{U_{\mathsf{CS}}(\mathcal{R})}$ only finitely often, contradicting well-foundedness of $\rhd_\mu$,
   - or there are infinitely many $\to_{U_{\mathsf{CS}}(\mathcal{R})}$-steps in ($\dagger$). But then we can construct a sequence $s_1 \to_{U_{\mathsf{CS}}(\mathcal{R})/\rhd_\mu} s_2 \to_{U_{\mathsf{CS}}(\mathcal{R})/\rhd_\mu} s_3 \to_{U_{\mathsf{CS}}(\mathcal{R})/\rhd_\mu} \dots$ with $s_1 = t_1$, contradicting the fact that all elements of $\mathcal{T}(\mathcal{F}, \mathcal{V})$ are $\to_{U_{\mathsf{CS}}(\mathcal{R})/\rhd_\mu}$-terminating.
2. Next we show $\succ = (\succ \cup \rhd)^+$. The direction $\succ \subseteq (\succ \cup \rhd)^+$ is obvious. For the other direction, $(\succ \cup \rhd)^+ \subseteq \succ$, assume we have $s\ (\succ \cup \rhd)^{n+1}\ t$. Then we proceed by induction on n. In the base case $s\ (\succ \cup \rhd)\ t$. If $s \succ t$ we are done. Otherwise, $s \rhd t$ and thus also $s \rhd_\mu t$ since $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and therefore $s \succ t$. In the step case $n = k + 1$ for some k, and $s\ (\succ \cup \rhd)\ u\ (\succ \cup \rhd)^k\ t$. Then we obtain $s \succ u$ by a similar case-analysis as in the base case. Moreover $u \succ t$ by induction hypothesis, and thus $s \succ t$.
3. Now we show that $\to_{\mathcal{R}} \subseteq \succ$. Assume $s \to_{\mathcal{R}} t$. Together with simulation completeness of $U_{\mathsf{CS}}(\mathcal{R})$, Theorem 3, we get $s \to^+_{U_{\mathsf{CS}}(\mathcal{R})} t$ which in turn implies $s \succ t$.
4. Finally, we show that if for all $\ell \to r \Leftarrow s_1 \approx t_1, \dots, s_n \approx t_n$ in $\mathcal{R}$, substitutions $\sigma \colon \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$, and $0 \leqslant i < n$, if $s_j\sigma \to^*_{\mathcal{R}} t_j\sigma$ for all $1 \leqslant j \leqslant i$ then $\ell\sigma \succ s_{i+1}\sigma$. We have the sequence

   $$\ell\sigma \to^+_{U_{\mathsf{CS}}(\mathcal{R})} U^\rho_{i+1}(s_{i+1}, \mathsf{v}(\ell), \mathsf{ev}(t_1, \dots, t_i))\sigma \rhd_\mu s_{i+1}\sigma$$

   using the definition of $U_{\mathsf{CS}}(\mathcal{R})$ together with simulation completeness (Theorem 3). But then also $\ell\sigma \succ s_{i+1}\sigma$ as wanted because $\ell\sigma, s_{i+1}\sigma \in \mathcal{T}(\mathcal{F}, \mathcal{V})$.

Hence $\mathcal{R}$ is quasi-decreasing with the order $\succ$. ◀

Table 1 (Non-)quasi-decreasing DCTRSs out of 103 in Cops by transformation and tool.

	conditional $\mathcal{R}$			$U_{\mathsf{CS}}(\mathcal{R})$			$U(\mathcal{R})$					
	AProVE	MU-TERM	VMTL	AProVE	MU-TERM	VMTL	AProVE	MU-TERM	NaTT	TTT2	VMTL	total
YES	80	78	80	78	78	79	81	78	77	78	78	84
NO	–	12	–	–	–	–	–	–	–	–	–	12

The converse of Theorem 5 has already been shown by Schernhammer and Gramlich [10, Theorem 4]:

▶ **Theorem 6.** *If a DCTRS $\mathcal{R}$ is quasi-decreasing then the CSRS $U_{\mathsf{CS}}(\mathcal{R})$ is μ-terminating on original terms.* ◀

Thus the desired equivalence follows as an easy corollary.

▶ **Corollary 7.** *Quasi-decreasingness of a DCTRS $\mathcal{R}$ is equivalent to μ-termination of the CSRS $U_{\mathsf{CS}}(\mathcal{R})$ on original terms.*

4 Experiments

In order to present up-to-date numbers for (non-)quasi-decreasingness we conducted experiments on the 103 DCTRSs contained in the confluence problems database using various automated termination tools. Of these, AProVE [4], MU-TERM 5.13 [1], and VMTL 1.3 [9] are able to directly show quasi-decreasingness and MU-TERM is the only tool that can show non-quasi-decreasingness [7]. AProVE, MU-TERM, and VMTL can also handle context-sensitive systems and we used them in combination with $U_{\mathsf{CS}}(\mathcal{R})$. Finally, we also ran the previous tools together with NaTT [11] and TTT2 1.16 [5] on $U(\mathcal{R})$. The results for a timeout of one minute are shown in Table 1. There are several points of notice. The most yes-instances (81) we get if we use AProVE together with $U(\mathcal{R})$. Interestingly, AProVE cannot show quasi-decreasingness of system 362 directly, although it succeeds (like all other tools besides NaTT) if provided with its unraveling. Moreover, systems 266, 278, and 279 can be shown to be quasi-decreasing by AProVE if we use $U(\mathcal{R})$ but not if we use $U_{\mathsf{CS}}(\mathcal{R})$ (even if we increase the timeout to 5 minutes). On system 363 only MU-TERM succeeds (in the direct approach). If we compare MU-TERM on conditional systems to MU-TERM with $U_{\mathsf{CS}}(\mathcal{R})$, the direct method succeeds on system 360 but not on system 329. Conversely, when using $U_{\mathsf{CS}}(\mathcal{R})$ it succeeds on system 329 but not on system 360. Moreover, MU-TERM seems to have some problems with systems 278 and 342, generating errors in the direct approach. With $U_{\mathsf{CS}}(\mathcal{R})$ VMTL succeeds on 79 systems, subsuming the results from AProVE and MU-TERM (78 each). On system 357 only VMTL together with $U_{\mathsf{CS}}(\mathcal{R})$ succeeds. With $U(\mathcal{R})$, NaTT succeeds on 77 systems, this is subsumed by TTT2, succeeding on 78 systems, which in turn is subsumed by AProVE, succeeding, as mentioned above, on 81 systems. In total 84 systems are shown to be quasi-decreasing, 12 systems to be non-quasi-decreasing, and only 7 remain open. One of these, for example, is system 337 from Cops, for computing Bubble-sort [12]

$$x < 0 \to \mathsf{false} \qquad 0 < \mathsf{s}(y) \to \mathsf{true}$$
$$\mathsf{s}(x) < \mathsf{s}(y) \to x < y \qquad x : y : ys \to y : x : ys \Leftarrow x < y \approx \mathsf{true}$$

whose unraveling replaces the last (and only conditional) rule by the two rules:

$$x : y : ys \to \mathsf{U}(x < y, x, y, ys) \qquad \mathsf{U}(\mathsf{true}, x, y, ys) \to y : x : ys$$

5 Conclusion

We provide a direct proof for one direction of a previous characterization of quasi-decreasingness, i.e., that μ-termination of a CSRS $U_{\mathsf{CS}}(\mathcal{R})$ on original terms implies quasi-decreasingness of the DCTRS $\mathcal{R}$ without the need of a detour by using the notion of context-sensitive quasi-reductivity. We believe that our proof could easily be adapted to any other context-sensitive transformation as long as it is simulation complete. Moreover, we provide experimental results on a recent collection of DCTRSs. Knowing that a DCTRS is quasi-decreasing is, among other things, useful to show confluence with the Knuth-Bendix criterion for CTRSs [2].

Acknowledgments. We thank the Austrian Science Fund (FWF project P27502) for supporting our work. Moreover we would like to thank the anonymous reviewers for useful hints and remarks and particularly for pointing out a flaw in an earlier version of Section 4.

References

1 B. Alarcón, R. Gutiérrez, S. Lucas, and R. Navarro-Marset. Proving Termination Properties with MU-TERM. In *Proc. 13th AMAST*, volume 6486 of *LNCS*, pages 201–208. Springer Berlin Heidelberg, 2010. doi:10.1007/978-3-642-17796-5_12.

2 J. Avenhaus and C. Loría-Sáenz. On conditional rewrite systems with extra variables and deterministic logic programs. In *Proc. 5th LPAR*, volume 822 of *LNCS*, pages 215–229. Springer, 1994. doi:10.1007/3-540-58216-9_40.

3 F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

4 J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving Termination of Programs Automatically with AProVE. In *Proc. 7th IJCAR*, volume 8562 of *LNCS*, pages 184–191. Springer International Publishing, 2014. doi:10.1007/978-3-319-08587-6_13.

5 M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean termination tool 2. In *Proc. 20th RTA*, volume 5595 of *LNCS*, pages 295–304. Springer, 2009. doi:10.1007/978-3-642-02348-4_21.

6 S. Lucas. Context-sensitive computations in functional and functional logic programs. *J. Funct. Logic Progr.*, 1998(1), 1998.

7 S. Lucas, J. Meseguer, and R. Gutiérrez. Extending the 2D Dependency Pair Framework for Conditional Term Rewriting Systems. In *Proc. 24th LOPSTR*, volume 8981 of *LNCS*, pages 113–130. Springer International Publishing, 2014. doi:10.1007/978-3-319-17822-6_7.

8 E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002.

9 F. Schernhammer and B. Gramlich. VMTL - a modular termination laboratory. In *Proc. 20th RTA*, volume 5595 of *LNCS*, pages 285–294. Springer, 2009. doi:10.1007/978-3-642-02348-4_20.

10 F. Schernhammer and B. Gramlich. Characterizing and proving operational termination of deterministic conditional term rewriting systems. *J. Logic Algebr. Progr.*, 79(7):659–688, 2010. doi:10.1016/j.jlap.2009.08.001.

11 A. Yamada, K. Kusakari, and T. Sakabe. Nagoya Termination Tool. In *Proc. Joint 25th RTA and 12th TLCA*, volume 8560 of *LNCS*, pages 466–475. Springer-Verlag, 2014. doi:10.1007/978-3-319-08918-8_32.

12 T. F. Şerbănuţă and G. Roşu. Computationally equivalent elimination of conditions. In *Proc. 6th RTA*, volume 4098 of *LNCS*, pages 19–34. Springer-Verlag, 2006. doi:10.1007/11805618_3.

Non-termination of String and Cycle Rewriting by Automata

Hans Zantema[1,2] and Alexander Fedotov[1,2]

1 **Department of Computer Science, TU Eindhoven, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, email:** H.Zantema@tue.nl
2 **Institute for Computing and Information Sciences, Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands**

Abstract

At RTA 2015, Endrullis and Zantema [4] presented a technique to automatically prove non-termination of string rewriting and term rewriting, by finding an automaton for which the accepted language has properties from which non-termination can be concluded. Here we recapitulate this technique, and present some recent extensions to non-termination of cycle rewriting.

The basic idea is to find a non-empty regular language of terms that is closed under rewriting and does not contain normal forms. It is automated by representing the language by an automaton with a fixed number of states, and expressing the mentioned requirements in a SAT formula. Satisfiability of this formula implies non-termination. For cycle rewriting encoding the requirements shows up to be essentially harder than for sting rewriting; we deal with this by approximating them exploiting the notion of simulation.

Some preliminaries of this note are copied from [4].

1 Introduction

A basic approach for proving that a term rewriting system (TRS) is non-terminating is to prove that it admits a *loop*, that is, a reduction of the shape $t \to^+ C[t\sigma]$, see [6]. Indeed, such a loop gives rise to an infinite reduction $t \to^+ C[t\sigma] \to^+ C[(C[t\sigma])\sigma] \to \cdots$ in which in every step t is replaced by $C[t\sigma]$. In trying to prove non-termination, several tools ([1, 2]) search for a loop. An extension from [3], implemented in [1] searches for other explicit infinite reductions. Here we follow a completely different approach: non-termination immediately follows from the existence of a non-empty set of terms / strings that is closed under rewriting and does not contain normal forms. Our approach is to find such a set being the language accepted by a finite automaton, obtained from the satisfying assignment of a SAT formula describing the above requirements. Here we focus on string rewriting; [4] shows how this extends to term rewriting by using tree automata, and non-termination of systems like the S-rule from combinatory logic can be proved to be non-terminating fully automatically.

Hence the goal is to describe the requirements, namely non-emptiness, closedness under rewriting, and not containing normal forms, in a SAT formula.

As an example consider

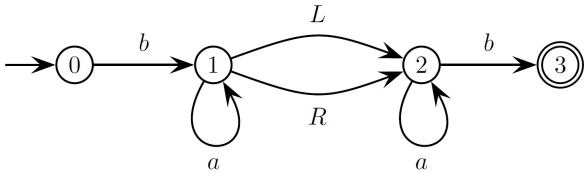
$$bL \to bR,\ Ra \to aR,\ Rb \to Lab,\ aL \to La.$$

Here we may choose the language described by the regular expression

$$b\, a^*\, (L + R)\, a^*\, b$$

also described by the automaton

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 13; pp. 13:1–13:5.



2 Abstract rewriting

An *abstract reduction system* (ARS) is a binary relation $\to$ on a set T. We write $\to^+$ for the transitive closure, and $\to^*$ for the reflexive, transitive closure of $\to$.

Let $\to$ be an ARS on T. The ARS $\to$ is called *terminating* or *strongly normalizing* (SN) if no infinite sequence $t_0, t_1, t_2, \ldots \in T$ exists such that $t_i \to t_{i+1}$ for all $i \geq 0$. A *normal form* with respect to $\to$ is an element $t \in T$ such that no $u \in T$ exists satisfying $t \to u$. The set of all normal forms with respect to $\to$ is denoted by $\mathsf{NF}(\to)$. The ARS $\to$ is called *weakly normalizing* (WN) if for every $t \in T$ a normal form $u \in T$ exists such that $t \to^* u$.

▶ **Definition 1.** A set $L \subseteq T$ is called

- *closed under* $\to$ if for all $t \in L$ and all $u \in T$ satisfying $t \to u$, it holds that $v \in L$, and
- *weakly closed under* $\to$ if for all $t \in L \setminus \mathsf{NF}(\to)$ there exists $u \in L$ such that $t \to^+ u$.

It is straightforward from these definitions that closed under $\to$ implies weakly closed under $\to$. The following theorems relate these notions to SN and WN, for the simple proofs we refer to [4].

▶ **Theorem 2.** *An ARS $\to$ on T is not* SN *if and only if a non-empty $L \subseteq T$ exists such that $L \cap \mathsf{NF}(\to) = \varnothing$ and L is weakly closed under $\to^+$.*

▶ **Theorem 3.** *An ARS $\to$ on T is not* WN *if and only if a non-empty $L \subseteq T$ exists such that $L \cap \mathsf{NF}(\to) = \varnothing$ and L is closed under $\to$.*

3 Encoding in SAT for string rewriting

First we focus on string rewriting: a string rewrite system (SRS) over Σ is defined to be a subset R of $\Sigma^* \times \Sigma^*$. Elements (ℓ, r) of an SRS are called (string rewrite) rules and are usually written as $\ell \to r$, where ℓ is called the left hand side (lhs) and r the right hand side (rhs) of the rule. The string rewrite relation $\to_R$ on Σ^* is defined by $u \to_R v \iff \exists x, y \in \Sigma^*, \ell \to r \in R : u = x\ell y \wedge v = xry$.

For encoding the requirements in Theorem 2 and Theorem 3 in a SAT formula, we fix a number n to be the number of states of the automaton M we are looking for. We introduce mn^2 boolean variables v_{ija} describing whether there is an a-transition from state i to state j, for i, j running over all n states, and a running over all m symbols.

The first requirement is $\mathcal{L}(M) \neq \varnothing$. This is expressed by stating that there is a path in M from the initial state to a final state of length $\leq 2^k$ for $k = \lceil \log_2 n \rceil$. In its turn this is expressed by additional variables p_{ijq} expressing that there is a path of length $\leq 2^q$ from i to j, for $q = 0, \ldots, k$.

The second requirement is closed under rewriting. One way to deal with this is to use the following over-approximation: for every rule $\ell \to r$ and every two states i, j in A for which there is a path labeled by ℓ from i to j, there is also a path labeled by r from i to j.

The last requirement is that $\mathcal{L}(M)$ contains no normal forms. One way to deal with this is the following. In a preprocessing build an automaton M' exactly accepting the normal

forms, then in the product automaton $M \times M'$ there should be no path from the initial state to a state (f, f') for which f is final in M and f' is final in M'.

For string rewriting this approach works well: for several examples non-termination can be proved automatically in this way where earlier techniques fail, see [4].

4 Cycle rewriting

For a string rewriting system R over Σ, apart from considering its string rewrite relation $\to_R$, we can also consider its cycle rewrite relation $\circlearrowright_R$ on cycles. Here a cycle is a string in which the start and end are connected, more precisely, a string modulo $\sim$ for $\sim$ defined by

$$u \sim v \iff \exists u_1, u_2 \in \Sigma^* : u = u_1u_2 \wedge v = u_2u_1.$$

For an SRS R over Σ the corresponding cycle rewrite relation $\circlearrowright_R$ on equivalence classes of $\sim$ is defined as follows:

$$[u] \circlearrowright_R [v] \iff \exists x \in \Sigma^*, \ell \to r \in R : \ell x \sim u \wedge rx \sim v.$$

Equivalently, one can state $[u] \circlearrowright_R [v] \iff \exists u' \in [u], v' \in [v] : u' \to_R v'$.

Termination of $\circlearrowright_R$ is a strictly stronger property than termination of $\to_R$, for instance for R consisting of the single rule $ab \to ba$ the string rewrite relation $\to_R$ is clearly terminating, but since $ab \sim ba$ the cycle rewrite relation $\circlearrowright_R$ is not terminating. Termination of cycle rewriting has been studied in [8, 7]. Here we wonder whether Theorem 2 and Theorem 3 can be applied to automatically prove non-termination of cycle rewriting. Unavoidably, the expression in SAT will be more complicated. For instance, for closedness under rewriting we do not need to conclude $urv \in \mathcal{L}(M)$ from $u\ell v \in \mathcal{L}(M)$ for $\ell \to r \in R$, but also allow $r_2vur_1 \in \mathcal{L}(M)$ if $r = r_1r_2$.

Focusing on Theorem 3 there are two ways to deal with this. We can look for an automaton M such that $\mathcal{L}(M)$ is closed under cyclic shift, that is, if $u \sim v$ then $u \in \mathcal{L}(M) \iff v \in \mathcal{L}(M)$, and require that the conditions of Theorem 3 hold for $\mathcal{L}(M)$, or we look for an automaton M such that $\mathcal{L} = \{u \mid \exists v \in \mathcal{L}(M) \wedge v \sim u\}$ satisfies the conditions of Theorem 3. Experiments show that the latter is the more powerful. More precisely, the following theorem is exploited.

▶ **Theorem 4.** *Let R be an SRS over Σ and $L_{NF} = \{u \mid [u] \in NF(\circlearrowright_R)\}$. Then $\circlearrowright_R$ is not WN if there exists $L \subseteq \Sigma^*$, such that:*

(1) $L \neq \varnothing$,
(2) for all $\ell \to r \in R$ and for all $u, v \in \Sigma^*$ it holds that if $u\ell v \in L$ then either $urv \in L$ or there exist $r_1 \neq \epsilon \neq r_2$ such that $r = r_1r_2$ and $r_2vur_1 \in L$,
(3) for all $\ell_1\ell_2 \to r \in R$ and for all $u \in \Sigma^*$ it holds that if $\ell_2u\ell_1 \in L$, then there exist r_1, r_2 (possibly empty) such that $r = r_1r_2$ and $r_2ur_1 \in L$,
(4) $L \cap L_{NF} = \varnothing$.

The (straightforward) proof is in [5]. The approach is to find an automaton M such that $L = \mathcal{L}(M)$ satisfies the requirements of Theorem 4. The main issue is how to express these requirements in SAT, or properties from which these requirements follow. The key idea is to exploit the notion of a *simulation*.

A relation S on the states of an automaton is called a *(forward) simulation* if for every p, q, r, a such that pSq and $p \xrightarrow{a} r$, there exists a state s such that $q \xrightarrow{a} s$ and rSs. A consequence of this definition is that for any string u if there is a u-path from p to r and

pSq holds, then there exists a state s such that rSs holds and there is a u-path from q to s. A backward simulation is the same but then for the reversed relation $\xrightarrow{a}$.

For the SAT encoding for every state t we introduce a forward simulation S_t and a backward simulation B_t, that is, we express each of these $2n$ new relations by n^2 fresh boolean variables, and add requirements expressing the above definitions. Moreover, we require for every t that if $(q_0, q) \in B_t$ then $q = t$, and if $(q_f, q) \in S_t$ then $q = t$, for all states q, t and the initial state q_0 and the single final state q_f. Write $p \xrightarrow{u} q$ if there is a u path from p to q, for any states p, q, $u \in \Sigma^*$. Using these properties one derives:

- if $(p, q) \in B_t$ and $q_0 \xrightarrow{u} p$, then $t \xrightarrow{u} q$.
- if $(p, q) \in S_t$ and $p \xrightarrow{u} q_f$, then $q \xrightarrow{u} t$.

Exploiting this, the part $u\ell v \in L \Rightarrow r_2 v u r_1 \in L$ from Theorem 4 (2), can be expressed by

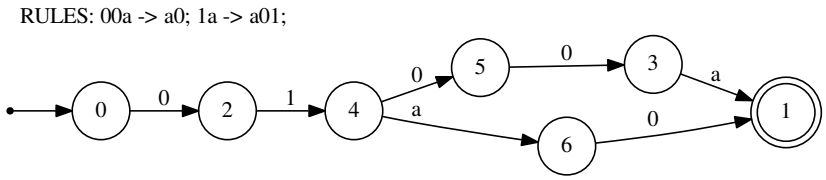
$$\forall p, q : p \xrightarrow{\ell} q \Rightarrow \exists p', q', t : q_0 \xrightarrow{r_2} p' \wedge q' \xrightarrow{r_1} q_f \wedge (p, q') \in B_t \wedge (q, p') \in S_t.$$

Combined with the other part of Theorem 4 (2) and the right quantification over r_1, r_2, this can be expressed in a SAT formula in a standard way.

Theorem 4 (3) is expressed in a similar way; Theorem 4 (1) is encoded similarly as for string rewriting.

For the remaining part Theorem 4 (4) in a preprocessing phase we compute an automaton M' which accepts the complement of L_{NF}. This is slightly harder than the similar condition for string rewriting. Let q_0' be the initial state and q_f' be the single final state of M'. Then for Theorem 4 (4) we require that there is a forward simulation relation $S = S_{q_f'}$ on $M \cup M'$ satisfying $(q_0, q_0') \in S$ and for which $q = q_f'$ is the only state satisfying $(q_f, q) \in S$. From these properties follows $\mathcal{L}(M) \subseteq \mathcal{L}(M')$, being part Theorem 4 (4). For more details and proofs we refer to [5]. Note that this encoding of $L \cap L_{\mathrm{NF}} = \varnothing$ gives rise to much smaller formulas than the encoding based on product automata as presented in [4].

We conclude by an example. Consider the SRS $R = \{00a \to a0, 1a \to a01\}$. Applying a SAT solver to the encoding as sketched above for this example yields a satisfying assignment from which the following automaton can be obtained, being a certificate for non-weak-normalization by Theorem 4.



Indeed, the requirements of Theorem 4 can be checked by hand, exploiting the properties of the simulations:

1. We verify that $L \neq \varnothing$ by checking that the final state 1 is reachable from the initial state 0 and we observe that this is the case.
2. We verify, that each occurrence of $u00av$ in L implies that either $ua0v \in L$, or $0vua \in L$, for any $u, v \in \Sigma^*$. Similarly, each $u1av$ in L implies that either $ua01v \in L$, or $01vua \in L$, or $1vua0 \in L$. $01\mathbf{00a} \in L$. Here, $u = 01$, $v = \varepsilon$. This is covered by $01\mathbf{a0} \in L$, where $u = 01$ and $v = \varepsilon$. Next, we have a $0\mathbf{1a}0 \in L$, where $u = 0$ and $v = 0$. This is covered

by **01**00**a** $\in L$, where $u = 0$ and $v = 0$. $(6, 4) \in S_5$: a 0-transition from 6 to 1 can be mimicked by an 0-transition from 4 to 5, 1 has no outgoing transitions and $(1, 5) \in S_5$. $(2, 3) \in B_5$: an incoming 0-transition from 0 to 2 can be mimicked by an incoming 0-transition from 5 to 3, 0 has no incoming transitions and $(0, 5) \in B_5$.

3. We verify, that for each occurrence of either $0au0$, or $au00$ in L, either $ua0 \in L$, or $a0u \in L$, or $0ua \in L$, for any $u \in \Sigma^*$. For each $au1 \in L$, there must be either $ua01 \in L$, or $a01u \in L$, or $01ua \in L$, or $1ua0 \in L$, for any $u \in \Sigma^*$. This holds, since none of $0au0$, or $au00$, or $au1$ is present in L.
4. We verify that $L \cap L_{\mathrm{NF}} = \varnothing$. This is the case, since every accepting path contains either 01**00a**, or 01**a0**.

References

1 Homepage of AProVE, 2015. `http://aprove.informatik.rwth-aachen.de`.

2 Homepage of TTT2, 2015. `http://cl-informatik.uibk.ac.at/software/ttt2/`.

3 F. Emmes, T. Enger, and J. Giesl. Proving Non-looping Non-termination Automatically. In *International Joint Conference on Automated Reasoning (IJCAR 2012)*, volume 7364 of *LNCS*, pages 225–240. Springer, 2012.

4 J. Endrullis and H. Zantema. Proving non-termination by finite automata. In Maribel Fernandez, editor, *26th International Conference on Rewriting Techniques and Applications (RTA'15)*, volume 36 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 160–176, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

5 A. Fedotov. Proving non-termination of cycle rewriting using automata. Master's thesis, Radboud University Nijmegen, `http://www.win.tue.nl/~hzantema/fedotov.pdf`, 2016.

6 A. Geser and H. Zantema. Non-looping String Rewriting. *RAIRO Theoretical Informatics and Applications*, 33(3):279–302, 1999.

7 D. Sabel and H. Zantema. Transforming cycle rewriting into string rewriting. In Maribel Fernandez, editor, *26th International Conference on Rewriting Techniques and Applications (RTA'15)*, volume 36 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 285–300, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

8 H. Zantema, H.J.S. Bruggink, and B. Koenig. Termination of cycle rewriting. In G. Dowek, editor, *Proceedings of Joint International Conference, RTA-TLCA 2014*, volume 8560 of *Lecture Notes in Computer Science*, pages 476–490. Springer, 2014.

Termination of Term Graph Rewriting

Hans Zantema[1,2], Dennis Nolte[3], and Barbara König[3]

1 **TU Eindhoven** – h.zantema@tue.nl
2 **Radboud University Nijmegen**
3 **Duisburg Essen University** – {barbara_koenig,dennis.nolte}@uni-due.de

1 Introduction

In implementing term rewriting, an obvious optimization is to share common subterms. In this way the objects to rewrite are not terms represented by trees, but by directed graphs. For finite terms the directed graphs will be acyclic, but in many applications, in particular in functional programming ([7]), it makes sense to also allow cycles, by which after unfolding the represented term is infinite. In the graph every node is labeled by an operation symbol, and the outgoing edges of such a node are numbered from 1 to the arity of the operation symbol. These graphs are called *term graphs*, and rewriting on term graphs has been extensively studied, see e.g. [8, 4], sometimes under the name of jungle rewriting [6, 5]. In this note we focus on term graphs that are not required to have a root. The key issues of this note are the following.

- How to interpret a (left-linear non-collapsing) term rewrite rule when applied to term graphs? We argue that there are two natural ways to do so. One is the *extended version*, coinciding with the version as studied before, e.g., in [4], covering implicit unraveling. In contrast, in the *basic version*, implicit unraveling is not allowed. One motivation of this note was to generalize cycle rewriting as studied in [11, 9], being string rewriting applied on cycles, to term rewriting applied on graphs, and then this basic version is the natural notion where the single rule $f(g(x)) \to f(x)$ is terminating, but $f(g(x)) \to g(f(x))$ is not.
- How to prove termination of term graph rewriting automatically? We observed that conceptually there is a strong relationship between term graph rewriting and graph transformation systems (GTSs), and in [3, 2] sophisticated techniques have been developed to prove termination of GTSs automatically, often based on SMT solving, and implemented in the tool *Grez*. So our approach is to transform term graph rewriting to graph transformation, in such a way that termination of the term graph rewrite system (TGRS) can be concluded from termination of the resulting GTS, to be proved by *Grez*.

For termination issues an extensive database of benchmarks has been developed: TPDB [1], including a great number of term rewriting systems (TRSs). So for testing our approach, it is natural to filter a suitable selection from this database. For applying them on term graphs, the TRSs should be left-linear and non-collapsing, since otherwise their semantics remains unclear. For a selection of 201 TRSs obtained by a filtering guided by these observations, we applied *Grez*.

In this note most details are omitted; for the full version we refer to [12].

2 Term graph rewriting

▶ **Definition 1.** A term graph over a signature Σ is a triple $(V, \mathsf{lab}, \mathsf{succ})$ in which

- V is a finite set of nodes (vertices),
- $\mathsf{lab} : V \to \Sigma$ is a partial function, called labeling, and

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 14; pp. 14:1–14:5.

- $\mathsf{succ} : V \to V^*$ is a partial function, called successor, having the same domain as lab, such that for every $v \in V$ for which $\mathsf{succ}(v)$ is defined, the length of $\mathsf{succ}(v)$ is equal to the arity of $\mathsf{lab}(v)$: $|\mathsf{succ}(v)| = \mathsf{ar}(\mathsf{lab}(v))$.

This definition coincides with the one given in [4]. If $\mathsf{succ}(v) = (v_1, \ldots, v_n)$ then we see $(v, v_1), \ldots, (v, v_n)$ as the n outgoing edges of v. Note that in contrast to many other variants of graphs, here outgoing edges are ordered: swapping two outgoing edges changes the term graph. Term graphs are a direct extension of finite terms.

Just as in term rewriting, a term graph transformation rule consists of a left-hand side and a right-hand side, and the basic idea is that an occurrence of a left-hand side may be replaced by the corresponding right-hand side. Now left-hand sides and right-hand sides are term graphs themselves, and an occurrence of a left-hand side may be defined as an injective morphism from the left-hand side to the term graph to be rewritten. However, for a precise description some extra information is required: which nodes of the left-hand side correspond to nodes in the right-hand side, and what to do with the remainder of the left-hand side.

A *production* is defined to consist of two injective morphisms $\ell : I \to L$ and $r : I \to R$, where L is the *left-hand side*, R is the *right-hand side* and the term graph I is the *interface*. For nodes in L and R representing variables, lab and succ are undefined, and for nodes in I they may be undefined as well. The operational effect of a corresponding transformation of a term graph G is that an injective morphism from L to G is found, the part $\ell(I)$ of L is maintained while the rest of L (including labels not present in I) is removed, and nodes and edges from R that are not in $r(I)$ are added. This means that a production can only be applied if the graph nodes corresponding to nodes in $L \setminus \ell(I)$ have no other incoming edges than those that occur in L (so-called *dangling edge condition*). As the outgoing edges correspond to succ and the labels coincide, a similar requirement for outgoing edges always holds implicitly. A standard way to describe this effect of productions in a more abstract way is by the *double-pushout approach*.

We define a term graph rewrite system (TGRS) to be a set of (term graph) productions.

3 Interpret Term Rewriting in Term Graph Rewriting

We will focus on left-linear (left-hand sides of all rules are linear) and non-collapsing (right-hand sides of rules are no variables) TRSs and investigate natural ways to interpret them as TGRSs. They have to be non-collapsing, since we restrict to injective morphisms in the rule.

In order to apply a term rewriting rule, that is, a rule $t \to u$ in which t and u are finite terms, to a term graph, there are two natural ways to proceed. In both ways in the corresponding production $L \leftarrow_\ell I \to_r R$, the term graph L is the term graph of t. The right-hand side corresponds to the term graph of u, where nodes for every variable that occurs in t, but not in u, are added. The main difference is in the interface I: roughly speaking in the *basic* version it is as small as possible, while in the *extended* version it is nearly a full copy of L.

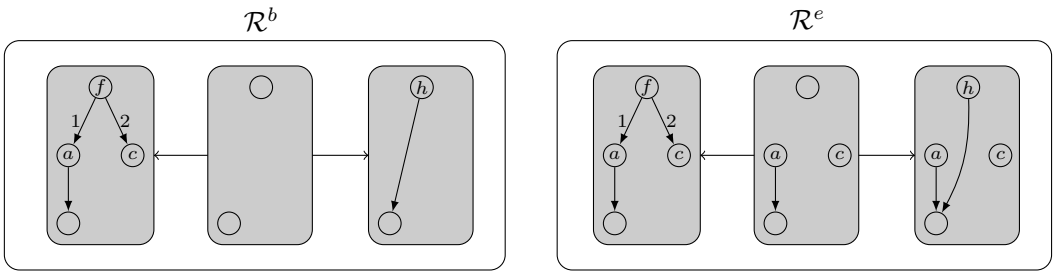
To motivate and define the two versions, let us first investigate what is really needed. The basic idea is that a part of the graph to be rewritten coincides with t, and that this is replaced by u. For doing so, the interface should at least contain the root of t and the variables of t. So in the *basic* version we define I to consist of the nodes of the term graph L of t that correspond to the root of t and to the variables in t, and no edges. The mappings ℓ, r map each of these nodes to the corresponding copy in L respectively R. The root is mapped to the root, and every node in I corresponding to a variable in t (u) is mapped to the corresponding node in L (R).

For string rewriting, that is, term rewriting in which all symbols are unary, termination of *cycle rewriting* as studied in [11, 9] coincides with termination of TGRSs in the basic version; the argument that symbols of other arity, not occurring in the rewrite system, do not influence the termination property, is similar to the argument given in [3].

The other version, the *extended* version, exactly corresponds to the version as presented in [4], where it is shown that for orthogonal TRSs, there is a correspondence between term graph rewriting and term rewriting on the corresponding unraveled (possibly infinite) terms (adequacy). Here the interface I is a copy of L, in which only the outgoing edges from the root are removed, that is, lab and succ are undefined for the node corresponding to the root of t. For the rest, I is a copy of L in which for all nodes succ and lab is defined, and by $\ell : I \to L$ every node and edge is mapped to itself. The right-hand side R is the union of $\mathsf{TG}(u)$ with the interface I. Finally, the morphism $r : I \to R$ maps every node or edge of the interface to the corresponding item in R.

For a TRS $\mathcal{R}$ we denote by $\mathcal{R}^b$ ($\mathcal{R}^e$) the corresponding TGRS in the basic (extended) version.

▶ **Example 2.** We interpret the TRS $\mathcal{R} = \{\rho\}$ with $\rho = f(a(x), c) \to h(x)$. Then $\mathcal{R}^b = \{\rho^b\}$ and $\mathcal{R}^e = \{\rho^e\}$ are the corresponding sets of productions where the interface morphisms are denoted by the node positions:



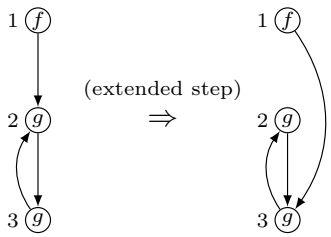
Termination (also called strong normalization) is abbreviated to SN. For a TRS $\mathcal{R}$ we write:

- $\mathsf{SN}(\mathcal{R})$ if $\mathcal{R}$ is terminating on finite terms,
- $\mathsf{SN}^b(\mathcal{R})$ if $\mathcal{R}^b$ is terminating on finite term graphs,
- $\mathsf{SN}^e(\mathcal{R})$ if $\mathcal{R}^e$ is terminating on finite term graphs.

It is not hard to prove $\mathsf{SN}^e(\mathcal{R}) \implies \mathsf{SN}^b(\mathcal{R}) \implies \mathsf{SN}(\mathcal{R})$ for all right-linear TRSs $\mathcal{R}$. The following example inspired by [10] shows that right-linearity is essential for the last implication. The TRS $f(0, 1, x) \to f(x, x, x)$, $a \to 0$, $a \to 1$ is non-terminating in term rewriting as $f(0, 1, a)$ rewrites in three steps to itself, but this cannot be mimicked in term graph rewriting without doing unraveling: our techniques easily prove SN^b.

For both implications the converse does not hold, as we will show now by examples. The single rule $f(g(x)) \to g(f(x))$ is the standard example of a string rewrite system that is terminating on strings but not on cycles, see [11], so this satisfies SN but not SN^b.

For the other implication, consider the single term rewrite rule $f(g(x)) \to f(x)$ and the term graph depicted in the left part of the picture to the right. We have three nodes $1, 2, 3$ with $\mathsf{lab}(1) = f$, $\mathsf{lab}(2) = \mathsf{lab}(3) = g$, and $\mathsf{succ}(1) = 2$, $\mathsf{succ}(2) = 3$ and $\mathsf{succ}(3) = 2$. In the extended version, an injective morphism from L to this graph is obtained by mapping the root of L to 1, and the two nodes below it to 2 and 3 respectively. By applying the



rule, the outgoing f-edge from 1 is removed, the rest of the left-hand side remains, and due to the right-hand side an edge from 1 to 3 is added, resulting in the graph depicted in the right part of the picture. As this graph is isomorphic to the original one, we see that this can be repeated forever, and the single rule $f(g(x)) \to f(x)$ does not satisfy SN^e.

In contrast, in the basic version the rule does not apply, since the middle node 2 has an incoming edge that is not in the left-hand side, and is not part of the interface. Hence, due to the dangling edge condition, the rule cannot be applied.

An elementary argument for proving SN and SN^b of the TRS $f(g(x)) \to f(x)$ is by counting the number of g's: in every step the number of g's strictly decreases. Hence our single rule $f(g(x)) \to f(x)$ does not satisfy SN^e but satisfies SN^b.

The remainder of this note is devoted to automatically proving termination of TRSs, interpreted as TGRSs in both versions: apply transformations to GTSs for which the tool *Grez* can be applied.

4 Transforming TGRSs to GTSs and experiments

The graphs in the GTSs on which *Grez* can be applied (shortly called graphs now) differ in three ways from term graphs: nodes may have any number of outgoing edges, these outgoing edges are not numbered, and the labels are not in the nodes but on the edges.

We propose two transformations from term graphs to graphs now, both having the property that an infinite reduction in term graph rewriting transforms to an infinite graph transformation reduction. For the number encoding, also the reverse is the case under mild conditions. For details we refer to [12].

The first transformation is called *function encoding*. The structure of the graph remains the same. The idea is that for a node labeled by a function symbol f of arity $n \geq 1$, the label of this node is removed, and the n ordered outgoing edges in the term graph are labeled by $f_1, \ldots, f_n$, respectively. In order to preserve constants, we introduce a fresh node $c(v)$, for every node $v \in V$ for which $\mathsf{lab}(v)$ is a constant.

The second transformation is called *number encoding*: then for a node v labeled by a symbol of arity $n > 1$, a fresh node is created, and an edge labeled by this symbol from v to this fresh node is added, while this fresh node has n outgoing edges labeled by $1, 2, \ldots, n$ to the nodes in $\mathsf{succ}(v)$.

Experiments

In the TPDB there is a folder *TRS Standard* consisting of 1498 TRSs. In our framework we are interested in non-collapsing left-linear TRSs. Only 621 of the TRSs are both left-linear and non-collapsing. We discarded 386 of these TRSs that exceeded tractability for the resulting GTSs. Another 34 TRSs were left out since they were obviously non-terminating. Of the 201 remaining TRSs 95 are right-linear.

We ran *Grez* on all remaining 201 examples using a Windows workstation with a 2,67 Ghz, 4-core CPU and 8 GB RAM. We used the weighted type graph technique over ordered semirings [2] and tried to find weighted type graphs which consist of 2 nodes. For all GTSs, where *Grez* could find a termination proof, the weighted type graphs were generated within a few seconds. Some TRSs satisfy SN but not SN^b due to cycles. Therefore, using the type graph technique, it is impossible to prove termination for these examples. To summarize the results, we present the following table:

Termination Analysis using *Grez* on TPDB (Standard)				
Left-linear + Non-Collapsing	621			
Too Many Rules (> 9)	-235			
Generated Graphs Too Large	-151			
Non-Terminating	-34			
Tested Total	**201**			
No Result Found	84			
Terminating GTS Total	**117**		**24**	
Terminating + Right-linear	**50**		**21**	
Terminating using	117 Number Encoding	115 Function Encoding	24 Number Encoding	24 Function Encoding
Version	Basic		Extended	

As a side effect, by applying our transformations to a selection of TRSs from the TPDB, we provided a substantial set of test cases for automatically proving termination of GTSs, which was lacking until now.

References

1 Termination problem database. `http://cl2-informatik.uibk.ac.at/mercurial.cgi/TPDB/file/d43b82fe816c/TRS_Standard`. [Online; accessed 21-January-2016].

2 H. J. S. Bruggink, B. König, D. Nolte, and H. Zantema. Proving termination of graph transformation systems using weighted type graphs over semirings. In *Graph Transformation–8th International Conference, ICGT 2015*, pages 52–68, 2015.

3 H.J.S. Bruggink, B. König, and H. Zantema. Termination analysis of graph transformation systems. In *Proc. of TCS 2014*, volume 8705 of *LNCS*. Springer, 2014.

4 A. Corradini and F. Drewes. Term graph rewriting and parallel term rewriting. In *Proceedings of the 6th International Workshop on Computing with Terms and Graphs (Termgraph)*, volume 48 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–18, 2011.

5 A. Corradini and F. Rossi. Hyperedge replacement jungle rewriting for term-rewriting systems and logic programming. *Theoretical Computer Science*, 109:7–48, 1993.

6 A. Habel, H.-J. Kreowski, and D. Plump. Jungle evaluation. In *Proc. Recent Trends in Data Type Specification*, volume 332 of *LNCS*, pages 92–112. Springer, 1988.

7 M. J. Plasmeijer and M. C. J. D. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley, 1993.

8 D. Plump. Term graph rewriting. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*. World Scientific, 1999.

9 D. Sabel and H. Zantema. Transforming cycle rewriting into string rewriting. In Maribel Fernandez, editor, *26th International Conference on Rewriting Techniques and Applications (RTA'15)*, volume 36 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 285–300, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

10 Y. Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25:141–143, 1987.

11 H. Zantema, H.J.S. Bruggink, and B. König. Termination of cycle rewriting. In G. Dowek, editor, *Proceedings of Joint International Conference, RTA-TLCA 2014*, volume 8560 of *LNCS*, pages 476–490. Springer, 2014.

12 Hans Zantema, Dennis Nolte, and Barbara König. Termination of term graph rewriting (full version), 2016. `www.win.tue.nl/~hzantema/ttg.pdf`.

Tool Papers

15th International Workshop on Termination (WST 2016).
Editors: Aart Middeldorp and René Thiemann

TcT: Tyrolean Complexity Tool

Martin Avanzini[1], Georg Moser[1], and Michael Schaper[1]

1 **Department of Computer Science, University of Innsbruck, Austria**
{martin.avanzini, georg.moser, michael.schaper}@uibk.ac.at

TcT is a fully automated complexity analyser supporting various formal systems and programming languages. Our tool is implemented in `Haskell`, open source, released under the BSD3 license, and available at

`http://cl-informatik.uibk.ac.at/software/tct` .

TcT does not make use of a *unique* problem representation, but employs a *variety* of different representations. The `tct-core` library implements an abstract *complexity framework* and complements it with a simple but powerful problem-independent *strategy language* that facilitates proof search. Problem specific techniques and search strategies are implemented in various *`tct`-modules*. For details, we refer to [2].

`tct-trs` provides analysis of (innermost) runtime and (innermost) derivational complexity of *term rewrite systems (TRSs)*. This module implements most of the known techniques, see [5] for an overview, and supports proof certification via `CeTA` [3].

`tct-its` provides complexity analysis of *integer transition systems (ITSs)* following the approach in [4].

To analyse a computer program TcT incorporates *complexity reflecting abstractions*, that is, the resource bound on the obtained abstract program reflects upon the resource usage of the input program.

`tct-hoca` incorporates an abstraction of higher-order functional programs to TRSs [1] and makes use of the `tct-trs` module to analyse the resulting problem.

`tct-jbc` provides a term-based abstraction of object-oriented bytecode programs to TRSs and ITSs [6].

Similarly we support *`C` integer programs* using existing (external) tools that abstract `C` programs to ITSs.

References

1 M. Avanzini, U. Dal Lago, and G. Moser. Analysing the Complexity of Functional Programs: Higher-Order Meets First-Order. In *Proc. 20th ICFP*. ACM, 2015. to appear.

2 M. Avanzini, G. Moser, and M. Schaper. TcT: Tyrolean Complexity Tool. In *Proc. 22th TACAS*, LNCS, pages 407–423, 2016.

3 M. Avanzini, C. Sternagel, and R. Thiemann. Certification of Complexity Proofs using CeTA. In *Proc. of 26th RTA*, volume 36 of *LIPIcs*, 2015.

4 M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Alternating Runtime and Size Complexity Analysis of Integer Programs. In *Proc. 20th TACAS*, LNCS, 2014.

5 G. Moser. Proof Theory at Work: Complexity Analysis of Term Rewrite Systems. *CoRR*, abs/0907.5527, 2009. Habilitation Thesis.

6 G. Moser and M. Schaper. A Complexity Preserving Transformation from Jinja Bytecode to Rewrite Systems. *CoRR, cs/PL/1204.1568*, 2012. last revision: May 6, 2014.

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 15; pp. 15:1–15:1.

VeryMax: Tool Description for termCOMP 2016

Cristina Borralleras[1], Daniel Larraz[2], Albert Oliveras[2], José Miguel Rivero[2], Enric Rodríguez-Carbonell[2], and Albert Rubio[2]

1 Universitat de Vic
2 Universitat Politècnica de Catalunya, Barcelona

VeryMax is a verification framework for checking automatically safety properties, and proving termination and non-termination of sequential programs with unbounded non-determinism. It heavily depends on Max-SMT constraint solving for doing program analysis efficiently [5].

The tool can handle, through LLVM, programs written in a small fragment of the C and C++ languages, which includes linear operations with integer variables[1], and loop and conditional statements. It also has partial support for boolean variables, and integer division and modulo operations. Besides, VeryMax can handle integer transition systems given in the T2 and the Pushdown SMTLIB2 specification formats.

In order to prove termination, VeryMax implements a new method [1] consisting in applying conditional termination and restricting step by step the states where non-termination may occur. The procedure for generating conditions for termination is a variation of the constraint-based approach for proving (unconditional) termination in [3], in combination with the ideas for generating preconditions for proving safety properties described in [2]. For each strongly connected subgraph (SCSG) of the control flow graph, a lexicographic conditional termination argument is iteratively constructed. This provides a precondition for ensuring termination, which is checked to be safe. In case of success termination is guaranteed. Otherwise, if the calculated precondition is not satisfied in some reachable states, the SCSG is *narrowed* filtering out the evaluations that are already known to be terminating, and a new termination argument is sought for the resulting SCSG. This method provides both a way of obtaining termination proofs by cases and a compositional approach to prove termination of larger code including sequences of loops.

In case of failing to prove termination, the resulting restricted set of states can be used as a starting point for a non-termination analysis. VeryMax disproves termination by generating conditional invariants that forbid executing any of the outgoing transitions that leave an SCSG, and then checking that the found conditional invariants are reachable [4].

The `VeryMax` tool is available at `www.cs.upc.edu/~albert/VeryMax.html`.

References

1 C. Borralleras, M. Brockschmidt, D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Proving termination through conditional termination. In *WST*, 2016.
2 M. Brockschmidt, D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Compositional Safety Verification with Max-SMT. In *FMCAD'15*, pp. 33-40, 2015.
3 D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Proving termination of imperative programs using Max-SMT. In *FMCAD'13*, pp. 218-225, 2013.
4 D. Larraz, K. Nimkar, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Proving non-termination using Max-SMT. In *CAV'14*, LNCS 8559, pp. 779-796. Springer, 2014.
5 D. Larraz, A. Oliveras, E. Rodríguez-Carbonell, and A. Rubio. Minimal-Model-Guided Approaches to Solving Polynomial Constraints and Extensions. In *SAT'14*, LNCS 8561, pp. 333-350. Springer.2014.

[1] As in many other tools, integers are treated as mathematical integers, so no overflows can occur.

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 16; pp. 16:1–16:1.

AProVE at the Termination Competition 2016

M. Brockschmidt[1], F. Frohn[2], C. Fuhs[3], J. Giesl[2], J. Hensel[2], D. Korzeniewski[2], M. Naaf[2], and T. Ströder[2]

1 **Microsoft Research, Cambridge, UK**
2 **LuFG Informatik 2, RWTH Aachen University, Germany**
3 **Birkbeck, University of London, UK**

AProVE is a tool for termination and complexity proofs of Java, C, Haskell, Prolog, and rewrite systems (possibly with built-in integers). To analyze programs, AProVE automatically converts them to (int-)TRSs. Then, numerous techniques are employed to prove termination and to infer complexity bounds for the resulting rewrite systems. The generated proofs can be exported to check their correctness using automatic certifiers. See [4] for an overview and the techniques implemented in AProVE. The following features were recently added:

***Decreasing Loops* for Lower Runtime Complexity Bounds of Term Rewriting** In [2], we presented an *induction technique* to deduce lower bounds for (worst-case) runtime complexity of TRSs. In the full version of [2], we now developed an alternative new technique, which searches for "*decreasing loops*". Decreasing loops generalize the notion of loops for TRSs and allow us to detect families of rewrite sequences with linear, exponential, or infinite length. While the power of our two techniques is orthogonal, loop detection is much more efficient and applicable to most examples.

Upper and Lower Runtime Complexity Bounds of Integer Transition Systems In [1], we improved our previous modular approach for complexity analysis of integer programs which alternates between finding symbolic time bounds for program parts and using these to infer bounds on the absolute values of program variables. Now our technique can also synthesize exponential bounds and we extended the modularity of our analysis such that program parts (e.g., library procedures) can be handled completely independently.
Moreover in [3], we developed a technique to infer *lower* bounds for the worst-case complexity of integer programs. It uses a framework for iterative, under-approximating program simplification and deduces asymptotic lower bounds from the resulting simplified programs. Our techniques to infer upper and lower bounds for integer transition systems are implemented in the tools KoAT and LoAT, which are integrated in AProVE.

Improvements in Termination Analysis of C Programs We recently extended our approach for automated termination analysis of C programs with explicit pointer arithmetic by a bit-precise modeling of bitvector integers [5]. Moreover, we improved AProVE's capabilities for proving non-termination of C programs.

References

1 M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Analyzing runtime and size complexity of integer programs. *ACM TOPLAS*, 38(4), 2016.
2 F. Frohn, J. Giesl, J. Hensel, C. Aschermann, and T. Ströder. Inferring lower bounds for runtime complexity. In *Proc. RTA '15*, LIPIcs 36, pages 334–349, 2015. Full version to appear in the *Journal of Automated Reasoning*.
3 F. Frohn, M. Naaf, J. Hensel, M. Brockschmidt, and J. Giesl. Lower runtime bounds for integer programs. In *Proc. IJCAR '16*, LNAI 9706, pages 550–567, 2016.
4 J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving termination of programs automatically with AProVE. In *Proc. IJCAR '14*, LNAI 8562, pages 184–191, 2014.
5 J. Hensel, J. Giesl, F. Frohn, and T. Ströder. Proving termination of programs with bitvector arithmetic by symbolic execution. *Proc. SEFM '16*, LNCS 9763, pages 234–252, 2016.

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 17; pp. 17:1–17:1.

CoFloCo: System Description

Antonio Flores-Montoya

TU Darmstadt, Dept. of Computer Science,
aeflores@cs.tu-darmstadt.de

CoFloCo is a cost analyis tool that infers upper and lower bounds on the resource consumption of programs expressed as cost relations. The tool is written in Prolog (compatible with SWI and YAP) and it uses the parma polyhedra library (PPL) [3]. CoFloCo is open source and it is available from `https://github.com/aeflores/CoFloCo`.

The input format of CoFloCo (Cost relations) can be generated from multiple languages: It has been integrated to be used as a backend in SACO [1] to analyze ABS programs; the COSTA tool [2] can generate cost relations from Java bytecode; and CoFloCo includes scripts to generate cost relations from KoAt's integer transition system format [4].

CoFloCo can be tried online through a web interface: `http://cofloco.se.informatik.tu-darmstadt.de`. In the web interface, it can be used alone to solve cost relation systems or combined with *llvm2kittel* (`https://github.com/s-falke/llvm2kittel`) to analyze single functions written in C.

The cost analysis has two phases. First a control-flow refinement of the cost relations is performed. This refinement breaks the (possibly complex) control-flow of the program into a set of simpler execution patterns. The second phase of the analysis computes the cost of such execution patterns incrementally using a special cost representation denoted cost structure. The technical details of this analysis can be found in [6, 5].

A unique feature of CoFloCo is the capability to compute piece-wise upper and lower bounds using the options `conditional_ubs` and `conditional_lbs`. This comes from the fact that CoFloCo computes a bound for each execution pattern detected in the control-flow refinement phase. Then, CoFloCo computes a precondition for each execution pattern and uses them to partition the input domain and provide a specialized bound for each partition. A detailed description of these and other options of CoFloCo can be found in its repository.

References

1 E. Albert, P. Arenas, A. Flores-Montoya, S. Genaim, M. Gómez-Zamalloa, E. Martin-Martin, G. Puebla, and G. Román-Díez. SACO: Static Analyzer for Concurrent Objects. In E. Ábrahám and K. Havelund, editors, *TACAS 2014*, volume 8413 of *Lecture Notes in Computer Science*, pages 562–567. Springer, 2014.

2 E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. *COSTA: Design and Implementation of a Cost and Termination Analyzer for Java Bytecode*, pages 113–132. Springer Berlin Heidelberg, 2008.

3 R. Bagnara, P. M. Hill, and E. Zaffanella. The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Quaderno 457, Dipartimento di Matematica, Università di Parma, Italy, 2006.

4 M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Analyzing runtime and size complexity of integer programs. *ACM Trans. Program. Lang. Syst.*, 38(4):13:1–13:50, Aug. 2016.

5 A. Flores-Montoya. Upper and lower amortized cost bounds of programs expressed as cost relations. In *21st International Symposium on Formal Methods (FM)*, 2016. ToAppear.

6 A. Flores-Montoya and R. Hähnle. Resource analysis of complex programs with cost equations. In J. Garrigue, editor, *APLAS*, volume 8858 of *Lecture Notes in Computer Science*, pages 275–295. Springer, Nov. 2014.

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 18; pp. 18:1–18:1.

MultumNonMulta: System Description

Dieter Hofbauer

ASW – Berufsakademie Saarland, Germany
d.hofbauer@asw-berufsakademie.de

MultumNonMulta aims at proving termination or non-termination of string rewriting systems automatically.

Its termination proofs are based on matrix interpretations as described in [6]. Various approaches for synthesizing matrix interpretations have been proposed, for instance complete enumeration of restricted matrix shapes, random guesses for small matrix dimensions, evolutionary programming, and, most prominently, constraint solving. MultumNonMulta uses a *backward completion procedure* as another approach for the same purpose, exploiting the view of matrix interpretations as weighted automata [5]. This is related to forward completion procedures for match-bound termination proofs as in [2]. Backward completion is easily adapted to the setting of relative termination proofs, thereby considerably strengthening its applicability. Building on results from [7], a specialized strategy supports automatically proving polynomial upper bounds on derivation lengths.

The current version of MultumNonMulta also comprises a loop-checker for proving non-termination. Formerly, this tool *KnockedForLoops* was described separately in [4, 8]. It implements a brute-force breadth-first enumeration of forward closures, based on the fact that the existence of a loop is equivalent to the existence of a looping forward closure [3]. As a simple combinatorial optimization, reductions are disregarded if there is a rewrite step to the left of the previous step and these steps do not overlap. Experimental comparisons between this approach and others can be found in [8] and [1].

References

1 F. Emmes, T. Enger, J. Giesl. *Proving non-looping non-termination automatically*. Proc. 6th Int. Joint Conf. on Automated Reasoning (IJCAR), Springer LNCS, Vol. 7364, pp. 225–240, 2012.

2 A. Geser, D. Hofbauer, J. Waldmann, H. Zantema. *Finding finite automata that certify termination of string rewriting*. Int. J. Found. Comput. Sci. 16(3):471–486, 2005.

3 A. Geser, H. Zantema. *Non-looping string rewriting*. Informatique Théorique et Applications 33(3):279–302, 1999.

4 D. Hofbauer. *System description: KnockedForLoops*. Proc. 13th Int. Workshop. on Termination (WST), p. 51, 2009.

5 D. Hofbauer. *Synthesizing Matrix Interpretations via Backward Completion*. Proc. 13th Int. Workshop. on Termination (WST), pp. 56–58, 2013.

6 D. Hofbauer, J. Waldmann. *Termination of string rewriting with matrix interpretations*. Proc. 17th Int. Conf. on Rewriting Techniques and Applications (RTA), Springer LNCS, Vol. 4098, pp. 328–342, 2006.

7 J. Waldmann. *Polynomially bounded matrix interpretations*. Proc. 21st Int. Conf. on Rewriting Techniques and Applications (RTA), pp. 357–372, 2010.

8 H. Zankl, C. Sternagel, D. Hofbauer, A. Middeldorp. *Finding and certifying loops*. Proc. 36th Int. Conf. on Current Trends in Theory and Practice of Computer Science (SOFSEM), Springer LNCS, Vol. 5901, pp. 755–766, 2010.

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 19; pp. 19:1–19:1.

CeTA – Certifying Termination and Complexity Proofs in 2016*

Sebastiaan J.C. Joosten, René Thiemann, and Akihisa Yamada

University of Innsbruck, Austria

CeTA is a certifier for automatically generated proofs. Its soundness – if CeTA accepts a proof of a certain property, then the property holds – is proven in the Isabelle/HOL [4] formalization IsaFoR [5]. A complete list of supported proof techniques as well as IsaFoR and CeTA itself are available at `http://cl-informatik.uibk.ac.at/software/ceta/`. We highlight some recent extensions of CeTA for validating complexity and termination proofs.

AC termination Previous versions of CeTA would model term rewrite systems modulo AC as relative rewrite systems and then apply techniques for relative rewriting. The new version now has support for AC dependency pairs [1], including refinements such as AC usable rules and AC dependency graphs [8].

Complexity of matrix interpretation CeTA can now precisely determine the asymptotic growth rate of A^n where A is the maximum-matrix determined by some matrix-interpretation [3]. To this end, algebraic numbers and Jordan-normal forms have been formalized [6, 7].

Integer transition systems Tools that prove termination or safety of imperative programs often abstract the program into an Integer Transition System (ITS).

Current CeTA can certify proofs of safety properties for ITSs. To show safety, a proof contains inductive invariants such that the error states have the inductive invariant False [2]. Together with Marc Brockschmidt, we work towards certifying termination proofs. These show that no transition can be taken infinitely often, using previously certified invariants.

References

1 K. Kusakari and Y. Toyama. On proving AC-termination by AC-dependency pairs. *IEICE T. Inf. Syst.*, E84-D(5):439–447, 2001.

2 K. McMillan. Lazy abstraction with interpolants. In *CAV'06*, volume 4144 of *LNCS*, pages 123–136, 2006.

3 G. Moser, A. Schnabl, and J. Waldmann. Complexity analysis of term rewriting based on matrix and context dependent interpretations. In *FSTTCS'08*, LIPIcs 2:304–315, 2008.

4 T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

5 R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *TPHOLs'09*, volume 5674 of *LNCS*, pages 452–468, 2009.

6 R. Thiemann and A. Yamada. Algebraic numbers in Isabelle/HOL. In *ITP'16*, volume 9807 of *LNCS*, pages 391–408, 2016.

7 R. Thiemann and A. Yamada. Formalizing Jordan normal forms in Isabelle/HOL. In *CPP'16*, pages 88–99. ACM, 2016.

8 A. Yamada, C. Sternagel, R. Thiemann, and K. Kusakari. AC dependency pairs revisited. In *CSL'16*, LIPIcs, 2016. To appear.

* This work was partially supported by FWF project Y757. The authors are listed in alphabetical order regardless of individual contributions or seniority.

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 20; pp. 20:1–20:1.

TermComp 2016 Partipicant: cycsrs 0.2*

David Sabel[1] and Hans Zantema[2,3]

1 **Goethe University Frankfurt am Main, Germany**
`sabel@ki.informatik.uni-frankfurt.de`
2 **TU Eindhoven, Department of Computer Science The Netherlands**
`h.zantema@tue.nl`
3 **Radboud University Nijmegen, Institute for Computing and Information Sciences, The Netherlands**

Our tool `cycsrs` aims to automatically prove cycle termination [5] of string rewrite systems. On the one hand it is a wrapper which allows to combine the use of termination tools, and on the other hand it performs sound and complete transformations on string termination problems, s.t. cycle termination coincides with string termination of the transformed system (see [4]). As an administrative task, `cycsrs` handles the calls to the different tools in succession on a given termination problem and the distribution of the time limit among the tools. Besides the termination provers AProVE [2] and T$_{\text{T}}$T$_2$ [3] and the Yices SMT-solver [1] which are used in the back-end, `cycsrs` uses three tools which prove cycle non-/termination without transformation:

- Torpacyc is a stand-alone tool for proving termination of cycle rewriting. It applies the techniques described in [5], in particular, weight interpretations, tropical and arctic matrix interpretations and match bounds. Moreover, also natural matrix interpretations as described in [4] are used.
- While Torpacyc targets the logic of unquantified linear integer arithmetic with uninterpreted sort and function symbols, the tool `TDMI` uses Yices to find matrix interpretations by targeting the logic of quantifier-free formulas over the theory of fixed-size bit vectors.
- The tool `Cycnt` performs a brute-force search to prove cycle non-termination.

Compared to version 0.1 which participated in TermComp 2015, version 0.2 of `cycsrs` includes the tool `TDMI` and it is now able to prove relative cycle termination using the tools `TDMI` and `Cycnt` and using new transformations to show relative cycle termination by proving relative string termination (using state-of-the-art termination provers).

Torpacyc can be downloaded from `http://www.win.tue.nl/~hzantema/torpa.html`, and the tools `cycsrs`, `TDMI`, and `Cycnt` are available as a single Haskell Cabal-package from `http://www.ki.informatik.uni-frankfurt.de/research/cycsrs/`.

References

1 B. Dutertre. Yices 2.2. In *Proc. CAV 2014*, *LNCS* 8559, pp. 737–744. Springer, 2014.
2 J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving termination of programs automatically with AProVE. In *Proc. IJCAR 2014*, *LNCS* 8562 , pp. 184–191. 2014.
3 M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean termination tool 2. In *Proc. RTA 2009*, *LNCS* 5595, pp. 295–304. Springer, 2009.
4 D. Sabel and H. Zantema. Transforming Cycle Rewriting into String Rewriting. In *RTA 2015*, LIPIcs 36, pp. 285–300, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015
5 H. Zantema, B. König, and H. J. S. Bruggink. Termination of cycle rewriting. In *Proc. RTATLCA 2014*, *LNCS* 8560, pp. 476–490. Springer, 2014.

* The first author is supported by the Deutsche Forschungsgemeinschaft (DFG) under grant SA 2908/3-1.

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 21; pp. 21:1–21:1.

Loopus – An Automatic Complexity Analyzer

Moritz Sinn

TU Wien
`sinn@forsyte.at`

Our tool `loopus` reads in the LLVM [3] intermediate representation and performs an intra-procedural analysis. It is capable of computing bounds for loops as well as analyzing the complexity of non-recursive functions. In both cases we use the *back-edge metric* as *cost model*, i.e., `loopus` infers a bound on the number of times that any back-jump instruction can be executed during program run.

`loopus` models integers as mathematical integers (not bit-vectors). We use the Z3 SMT solver [2] for performing control-flow refinement and program abstraction.

By default loopus soundly abstracts from all instructions which cannot directly be modeled in terms of integer valued expressions. Real code, however, often contains pointers and (recursive) data structures. A typical loop iteration pattern is the iteration over a *list* or a *tree*. As a means to test the potential of our tool and its performance and in order to find interesting examples, we implemented heuristic methods for handling non-integer code. These heuristics can be activated by command line parameters. If the corresponding command line parameter is set, `loopus` infers bounds on loops iterating over arrays or recursive data structures by introducing *shadow variables* that represent norms such as the length of a list or the size of an array. Further, `loopus` makes the following optimistic assumptions if the corresponding command line parameters are set: 1) pointers do not alias; 2) a recursive data structure is acyclic if a loop iterates over it; 3) a loop iterating over an array of characters is assumed to be terminating if an inequality check on the string termination character '\0' is found[1]; 4) given a loop condition of form '$a \neq 0$' `loopus` heuristically decides to either assume '$a > 0$' or '$a < 0$' as loop-invariant; 5) similarly, `loopus` assumes '$x > 0$' when an update of a loop counter of the form '$x = x * 2$' or '$x = x/2$' is detected. These assumptions are reported to the user if they were applied while computing the bound.

The bound algorithm implemented in loopus is described in [4] and [5]. Loopus is available for download at [1].

References

1 `http://forsyte.at/software/loopus/`.

2 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, pages 337–340, 2008.

3 Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, pages 75–88, 2004.

4 Moritz Sinn, Florian Zuleger, and Helmut Veith. A simple and scalable static analysis for bound analysis and amortized complexity analysis. In *CAV*, pages 745–761. Springer, 2014.

5 Moritz Sinn, Florian Zuleger, and Helmut Veith. Difference constraints: An adequate abstraction for complexity analysis of imperative programs. In *FMCAD*, pages 144–151, 2015.

[1] This assumption is often necessary to infer a bound since the type system of C does not distinguish between an array of characters and a string.

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 22; pp. 22:1–22:1.

$\mathsf{T_TT_2}$ @ TermComp'2016*

Christian Sternagel

University of Innsbruck, Austria
christian.sternagel@uibk.ac.at

The 2nd incarnation of the ***T**yrolean **T**ermination **T**ool* [1] is an automated tool for proving (and disproving) termination of term rewrite systems (TRSs) that has been developed in the *Computational Logic* group at the University of Innsbruck in Austria.

http://cl-informatik.uibk.ac.at/software/ttt2

Besides various minor changes and improvements, the most notable additions to version v1.16 of $\mathsf{T_TT_2}$ for this years termination competition are as follows.

Generalized Subterm Criterion. The previous SAT-based implementation of the subterm criterion is replaced by an SMT-based implementation [2] of the *generalized subterm criterion* due to Yamada et al. [3, Theorem 33].

▶ **Theorem.** *Let π be a multiprojection such that $\mathcal{P} \subseteq \trianglerighteq^{\pi}_{\mathsf{mul}}$ and $f(\ldots) \trianglerighteq^{\pi}_{\mathsf{mul}} r$ for all $f(\ldots) \to r \in \mathcal{R}$ with $\pi(f) \neq \varnothing$. Then $(\mathcal{P}, \mathcal{R})$ is finite iff $(\mathcal{P} \setminus \triangleright^{\pi}_{\mathsf{mul}}, \mathcal{R})$ is.* ◀

Generalized TCAP. Computing the *estimated dependency graph* now employs a generalization of tcap. First, given a TRS $\mathcal{R}$ over signature $\mathcal{F}$, let $\succ$ be the transitive closure of the relation $\{(f,g) \mid f(\ldots) \to g(\ldots) \in \mathcal{R}\} \cup \bigcup_{f(\ldots)\to x \in \mathcal{R}} \{(f,g) \mid g \in \mathcal{F}\}$ and note that $f \succeq g$ whenever $f(\ldots) \to^*_{\mathcal{R}} g(\ldots)$. Since the root symbols of non-variable terms are not changed by substitution $f \not\succeq g$ implies that there is no edge in the dependency graph from terms of the form $f(\ldots)$ to terms of the form $g(\ldots)$.[1] The generalized version of tcap in $\mathsf{T_TT_2}$ incorporates the information represented by $\succ$ and further makes use of non-linearity whenever possible. Consider Toyama's example $\mathsf{f}(x,\mathsf{a},\mathsf{b}) \to \mathsf{f}(x,x,x)$ whose dependency graph depends on whether $\mathsf{F}(x,x,x)\sigma \to^* \mathsf{F}(x,\mathsf{a},\mathsf{b})\tau$ for arbitrary substitutions σ and τ. However, this is only possible if $x\sigma \to^* \mathsf{a}$ *and* $x\sigma \to^* \mathsf{b}$, and thus requires that there is some $h \in \{\mathsf{f},\mathsf{a},\mathsf{b}\}$ such that $h \succeq \mathsf{a}$ *and* $h \succeq \mathsf{b}$. Since this is not the case, we obtain the empty dependency graph and thus may immediately conclude termination.

References

1 Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean Termination Tool 2. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications (RTA)*, volume 5595 of *Lecture Notes in Computer Science*, pages 295–304. Springer, 2009. doi:10.1007/978-3-642-02348-4_21.

2 Christian Sternagel. The generalized subterm criterion in $\mathsf{T_TT_2}$. In *Proceedings of the 15th Workshop on Termination (WST)*, 2016.

3 Akihisa Yamada, Christian Sternagel, René Thiemann, and Keiichirou Kusakari. AC dependency pairs revisited. In *Proceedings of the 25th EACSL Annual Conference on Computer Science Logic (CSL)*, Leibniz International Proceedings in Informatics, pages 84:1–84:16, 2016. to appear.

* The research described in this paper is supported by FWF (Austrian Science Fund) project P27502.

[1] This criterion for edge estimation came first up during private discussion with Akihisa Yamada and was first implemented in his tool NaTT.

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 23; pp. 23:1–23:1.

Matchbox at the 2016 Termination Competition

Johannes Waldmann[1]

1 **Fakultät IMN, HTWK Leipzig, Germany** johannes.waldmann@htwk-leipzig.de

matchbox [4] proves termination and non-termination of string rewriting and cycle rewriting. The design goal for 2016 is to have stand-alone implementation (pure Haskell code, without external constraint solvers) that is powerful for one-rule string rewriting, and for cycle-nontermination.

Termination is proved by matchbounds (for cycle rewriting), and RFC matchbounds (for string rewriting), using a recent re-implementation [3] of Endrullis' decomposition method [1].

Nontermination is proved by enumerating forward closures and transport systems, and checking for loops or rotations (for cycle rewriting).

A *rotating R-derivation* is a non-empty R-derivation $u \to_R^+ v$ such that there exist p, q with $u = qp, v = pq^+$. If R admits a rotating derivation, then R is cycle-nonterminating. We enumerate closures $u_0 \to^+ v_0$ and then determine w such that $u = u_0 w = qp$ and $v = v_0 w = pq^+$. This method was already used by matchbox in the 2015 competition.

A *transport system* [2] for R over Σ is given by (Γ, p, ϕ) where $\Gamma \subset \Sigma^+$ is finite, $p \in \Sigma^*$, and $\phi : \Gamma \to \Gamma^*$ is a morphism, such that $\forall x \in \Gamma : xp \to_R^* p\,\phi(x)$.

For example, $R = \{a0^3 \to 1^2a, b1^3 \to 0^5b\}$, constructed by Hans Zantema as test case nt15 for cycle termination, has, e.g., $ba0 \cdot 0^5 \to_R^3 0^5 \cdot b1a$, and thus admits a transport system

$$(\{0, b1a, ba0\}, 0^5, \{0 \mapsto 0, b1a \mapsto ba0 \cdot 0, ba0 \mapsto ba1\}).$$

A transport system is *rotating* if there is $x \in \Gamma$ with $x \in \phi^+(x)$. If R admits a rotating transport system, then R is cycle-nonterminating.

The transport system from the previous example is rotating for $x = b1a$, since $x \in \phi^2(x)$.

A transport system is *looping* if $p \in \Gamma^+$ and there are $x \in \Gamma, k \in \mathbb{N}$ such that $x^k p \sqsubseteq \phi^k(x)$, where $\sqsubseteq$ denotes the scattered subword relation. If R admits a looping transport system, then R is non-terminating. The point is that the length of the loop may be exponential in the size of the transport system.

Transport systems were used by matchbox in the 2008 competition, for string rewriting. The (obvious) variation (in fact, simplification) for cycle rewriting is new.

Before all of the above, matchbox tries to remove rules by additive weights. The corresponding system of linear inequalities is solved by Fourier Motzkin elimination. Also, matchbox strips common prefixes and suffixes from left and right hand sides of rules.

Pre-competition tests show that 1. matchbounds implementation performs well, 2. stripping and RFC matchbounds solve hard one-rule problems, and 3. rotating transport systems are quite helpful for cycle-nontermination.

References

1 Jörg Endrullis, Dieter Hofbauer, and Johannes Waldmann. Decomposing terminating rewrite relations. In *Workshop on Termination*, pages 39–43, 2006.

2 Johannes Waldmann. Compressed loops. http://www.imn.htwk-leipzig.de/~waldmann/matchbox/methods/loop.pdf, 2008.

3 Johannes Waldmann. On the construction of matchbound certificates. In *Termgraph*, 2016. https://gitlab.imn.htwk-leipzig.de/waldmann/pure-matchbox/tree/master/paper/matchbounds.

4 Johannes Waldmann. Pure matchbox. https://gitlab.imn.htwk-leipzig.de/waldmann/pure-matchbox, 2016.

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 24; pp. 24:1–24:1.

TermComp 2016 Participant: NaTT*

Akihisa Yamada

University of Innsbruck, Austria

1 Overview

NaTT [4], standing for Nagoya Termination Tool, or New alps Termination Tool,[1] is a termination prover for TRSs, which is available at

`http://www.trs.cm.is.nagoya-u.ac.jp/NaTT/`

Supported categories are as follows:

TRS/SRS Standard: NaTT implements only basic components of the *dependency pair (DP) framework* [2], and its power is mostly due to the *weighted path order* [5], which provides and strengthens many previously known reduction pair techniques as its instances. This year's version has a finer analysis of the dependency graph and usable rules.

TRS/SRS Relative: Since the last year NaTT is capable of proving relative termination via the DP framework [3].

TRS Equational: This year it implements a new formalized AC-DP framework [6].

All the reduction pair constraints are encoded into incremental SMT problem scripts, which can be piped to any solver that complies the SMT-LIB 2.0 standard. The competition version of NaTT uses Z3 [1] as the back-end SMT solver.

NaTT is particularly fast, due to several efforts in SMT encoding. Since state-of-the-art SMT solvers including Z3 are still not so efficient on non-linear problems, NaTT transforms non-linear expressions into linear ones using a straightforward but effective *if-then-else* blasting [4]. It moreover utilizes the incremental feature of SMT solvers, and this year it has been further optimized: generating variables and constraints corresponding to a rewrite rule (or a DP) is delayed until the rewrite rule is involved in the considered DP problem.

References

1 Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS 2008*, volume 4963 of *LNCS*, pages 337–340, 2008.

2 J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *LPAR 2004*, volume 3452 of *LNAI*, pages 75–90, 2004.

3 J. Iborra, N. Nishida, G. Vidal, and A. Yamada. Relative termination via dependency pairs. *J. Autom. Reasoning*, 2016. Available online.

4 A. Yamada, K. Kusakari, and T. Sakabe. Nagoya Termination Tool. In *RTA-TLCA 2014*, volume 8560 of *LNCS*, pages 466–475, 2014.

5 A. Yamada, K. Kusakari, and T. Sakabe. A unified order for termination proving. *Sci. Comput. Program.*, 111:110–134, 2015.

6 Akihisa Yamada, Christian Sternagel, René Thiemann, and Keiichirou Kusakari. AC dependency pairs revisited. In *CSL 2016*, LIPIcs. To appear.

* This research was partly supported by the Austrian Science Fund (FWF) project Y 757.

[1] Thanks to Masahiko Sakai for the original name, and Nao Hirokawa for the alternative.

Proceedings of the 15th International Workshop on Termination;
Editors: Aart Middeldorp and René Thiemann; Article No. 25; pp. 25:1–25:1.