17th International Workshop on Termination

WST 2021, July 16, 2021, affiliated with CADE-28 (Virtual event)
https://costa.fdi.ucm.es/wst2021/

Edited by

Samir Genaim

WST 2021 Proceedings

Preface

This report contains the proceedings of the 17th International Workshop on Termination, WST 2021, which was held virtually, July 16, 2021, affiliated with *the 28th International Conference on Automated Deduction* (CADE-28).

The Workshop on Termination traditionally brings 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 the different communities interested in termination (e.g., working on computational mechanisms, programming languages, software engineering, constraint solving, etc.). The friendly atmosphere enables fruitful exchanges leading to joint research and subsequent publications. The 17th International Workshop on Termination continues the successful workshops held in St. Andrews (1993), La Bresse (1995), Ede (1997), Dagstuhl (1999), Utrecht (2001), Valencia (2003), Aachen (2004), Seattle (2006), Paris (2007), Leipzig (2009), Edinburgh (2010), Obergurgl (2012), Bertinoro (2013), Vienna (2014), Obergurgl (2016), and Oxford (2018).

The WST 2021 program included an invited talk by Amir M. Ben-Amram on *Efficient Computation of Polynomial Resource Bounds for Bounded-Loop Programs*. WST 2021 received 12 submissions. After light reviewing the program committee decided to accept all submissions. The 12 contributions are contained in this proceedings.

I would like to thank the program committee members and the external reviewers for their dedication and effort, and the workshop and tutorial chair of CADE-28 for the invaluable help in the organization.

Madrid, July 2021 Samir Genaim

Organization

Program Committee

Martin Avanzini	INRIA Sophia, Antipolis
Carsten Fuhs	Birkbeck, U. of London
Samir Genaim (chair)	U. Complutense de Madrid
Jürgen Giesl	RWTH Aachen
Matthias Heizmann	U. of Freiburg
Cynthia Kop	Radboud U. Nijmegen
Salvador Lucas	U. Politécnica de Valéncia
Étienne Payet	U. de La Réunion
Albert Rubio	U. Complutense de Madrid
René Thiemann	U. of Innsbruck

External Reviewers

Didier Caucal
Enrique Martin-Martin
Fred Mesnard

Contents

Preface i

Organization iii

Invited Talks

Efficient Computation of Polynomial Resource Bounds for Bounded-Loop Programs
Amir M. Ben-Amram 1

Regular Papers

Did Turing Care of the Halting Problem?
Salvador Lucas 3

de Vrijer's Measure for SN of $\lambda^{\rightarrow}$ in Scheme
Nachum Dershowitz 9

Polynomial Loops: Termination and Beyond
Florian Frohn, Jürgen Giesl and Marcel Hark 15

Polynomial Termination over N is Undecidable
Fabian Mitterwallner and Aart Middeldorp 21

Modular Termination Analysis of C Programs
Frank Emrich, Jera Hensel and Jürgen Giesl 27

Loops for which Multiphase-Linear Ranking Functions are Sufficient
Amir Ben-Amram, Jesús J. Doménech and Samir Genaim 33

Analyzing Expected Runtimes of Probabilistic Integer Programs Using Expected Sizes
Fabian Meyer, Marcel Hark and Jürgen Giesl 39

Parallel Complexity of Term Rewriting Systems
Thaïs Baudon, Carsten Fuhs and Laure Gonnord 45

Between Derivational and Runtime Complexity
Carsten Fuhs 51

Mixed Base Rewriting for the Collatz Conjecture
Emre Yolcu, Scott Aaronson and Marijn Heule 57

Formalizing Higher-Order Termination in Coq
Deivid Do Vale and Niels van der Weide 63

Observing Loopingness
Étienne Payet 69

Efficient Computation of Polynomial Resource Bounds for Bounded-Loop Programs

Amir M. Ben-Amram ✉
School of Computer Science, The Tel-Aviv Academic College, Israel

Abstract

In Bound Analysis we are given a program and we seek functions that bound the growth of computed values, the running time, etc., in terms of input parameters. Problems of this type are uncomputable for ordinary, full languages, but may be solvable for weak languages, possibly representing an abstraction of a "real" program. In 2008, Jones, Kristiansen and I [3] showed that, for a weak but non-trivial imperative programming language, it is possible to decide whether values are polynomially bounded. In this talk, I will focus on current work [1, 2] with Geoff Hamilton, where we show how to compute tight polynomial bounds (up to constant factors) for the same language. We have been able to do that as efficiently as possible, in the sense that our solution has polynomial space complexity, and we showed the problem to be PSPACE-hard. The analysis is a kind of abstract interpreter which computes in a domain of symbolic bounds plus just enough information about data-flow to correctly analyse loops. A salient point is how we solved the problem for multivariate bounds through the intermediary of univariate bounds. Another one is that the algorithm looks for worst-case lower bounds, but a completeness result shows that the maximum of these lower-bound functions is also an upper bound (up to a constant factor).

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases Asymptotically-tight, Multivariate, Disjunctive, Worst-case, Polynomial bounds

References

1 Amir M. Ben-Amram and Geoff W. Hamilton. Tight worst-case bounds for polynomial loop programs. In Mikolaj Bojanczyk and Alex Simpson, editors, *Proceedings of the 22nd International Conference on Foundations of Software Science and Computation Structures, FOSSACS 2019*, volume 11425 of *Lecture Notes in Computer Science*, pages 80–97. Springer, 2019.

2 Amir M. Ben-Amram and Geoff W. Hamilton. Tight polynomial worst-case bounds for loop programs. *Log. Methods Comput. Sci.*, 16(2), 2020.

3 Amir M. Ben-Amram, Neil D. Jones, and Lars Kristiansen. Linear, polynomial or exponential? complexity inference in polynomial time. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Löwe, editors, *Proceedings of the 4th Conference on Computability in Europe, CiE 2008, Logic and Theory of Algorithms*, volume 5028 of *Lecture Notes in Computer Science*, pages 67–76. Springer, 2008.

Did Turing Care of the Halting Problem?

Salvador Lucas
DSIC & VRAIN, Universitat Politècnica de València, Spain, Spain

Abstract

The formulation and undecidability proof of the *halting problem* is usually attributed to Turing's 1936 landmark paper. In 2004, though, Copeland noticed that it was so named and, apparently, first stated in a 1958 book by Martin Davis. Indeed, in his paper Turing paid no attention to halting machines. Words (or prefixes) like "halt(ing)", "stop" or "terminat(e,ing)" do not occur in the text. Turing partitions his machines into two classes of *non-halting machines*, one of them able to produce the kind of real numbers he was interested in. His notion of computation did not require termination. His decidability results concerned the classification of his machines (*satisfactoriness* problem) and their 'productivity' (*printing* problem). No attempt to formulate or prove the halting problem is made. Other researchers were concerned with these issues, though. We briefly discuss their role in formulating what we currently understand as the *halting problem*.

2012 ACM Subject Classification Program analysis, Turing machines

Keywords and phrases Halting problem

Funding *Salvador Lucas*: Supported by projects RTI2018-094403-B-C32 and PROMETEO/2019/098.

1 Introduction

In 2016, I came to the idea that the 80^{th} anniversary of Turing's landmark paper (14) could be a good occasion to connect such a theoretical breakthrough and the birth of computers and Computer Science in some Spanish media for scientific dissemination. For this purpose, I started a thorough read of Turing's paper. As a member of the termination community, I was familiarized with the widespread idea that Turing proved the undecidability of the halting problem in this paper. So, I was shocked when, after finishing my reading, nothing similar to the halting problem showed up. Indeed, I then discovered that Copeland had already remarked the following (2, page 40):

> The halting problem was so named (and, it appears, first stated) by Martin Davis (4). Davis thinks it likely that he first used the term 'halting problem' in a series of lectures that he gave in 1952. (⋯) The proposition that the halting problem cannot be solved by computing machine is known as the 'halting theorem'. It is often said that Turing stated and proved the halting theorem in 'On Computable Numbers', but strictly this is not true.

I recently revisited the issue in (8). Its main arguments and conclusions are summarized below. Some new facts are also discussed, though.

2 Turing's notion of computability

Turing's notion of computation pays no attention to any halting behavior. In the subsection about *computing machines*, he writes (14, page 232):

> *if the machine is supplied with a blank tape and set in motion, starting from the correct initial m-configuration, the subsequence of the symbols printed by it which are*

of the first kind[1] *will be called the sequence computed by the machine.*

The machine is not required to halt, nor the subsequence of symbols of the "first kind" (the *figures*, i.e., 0, 1,...) is required to be finite. Then, the notions of *circular* and *circle-free* machines are introduced (14, page 233):

> *if a computing machine never writes down more than a finite number of symbols of the first kind, it will be called circular. Otherwise, it is called circle-free.*

Hence, circle-free machines *never* halt. The previous sentence may suggest that circular machines always halt. However, the clarification that immediately follows denies this:

> *A machine will be circular if it reaches a configuration from which there is no possible move, or it goes on moving, and possibly printing symbols of the second kind.*[2]

Examples of circular machines (given as programs)[3] are:

$\mathcal{P}_{C1}$: `skip`
$\mathcal{P}_{C2}$: `while true do print #`
$\mathcal{P}_{C3}$: `print 0; while true do skip`

Here, $\mathcal{P}_{C1}$ stops after doing nothing (first case of Turing's definition); $\mathcal{P}_{C2}$ prints infinitely many symbols of the second kind (second case); and $\mathcal{P}_{C3}$ prints `0` only once and then runs forever without printing anything else. An example of a circle-free machine is

$\mathcal{P}_{CF}$: `while true do print 0`

which prints infinitely many symbols of the first kind. Thus, *Turing partitions his machines into two classes, none of which is required to halt.* This suggests that, at least in (14), *he was not particularly interested in investigating halting machines.* Furthermore, he makes explicit that only infinite sequences of figures (computed by circle-free machines) are considered as *computable* (14, page 233):

> *A sequence is said to be computable if it can be computed by a circle-free machine.*

Circular machines may halt (e.g., $\mathcal{P}_{C1}$), but, by definition, circle-free machines *never stop.* Turing chooses the last ones to define his notion of computable sequence. Thus, *Turing's notion of computation did not rely on (actually* rejected*!) any termination requirement.*

3 Martin Davis' description of Turing machines and computations

The standard, algorithmic idea of a computation by a Turing Machine (TM) requires that the machine halts, thus defining a 'computed sequence' as the one which is obtained when the machine halts, cf. (4, Definition 1.9):

> *By a computation of a Turing machine $\mathcal{M}$ is meant a finite sequence of instantaneous descriptions $\alpha_1, \ldots, \alpha_p$ such that $\alpha_i \to \alpha_{i+1}$ for $1 \leq i < p$ and such that α_p is terminal with respect to $\mathcal{M}$.*

[1] Such symbols "of the first kind" are introduced on page 232: they are called *figures* and restricted to be either 0 or 1.
[2] Symbols "of the second kind" are also introduced on page 232 just as "the others", i.e., those which are not of the first kind. In this paper we only use #.
[3] To improve readability, for the informal examples of Turing machines we use a simple imperative language. This is a usual practice in the literature.

Davis' instantaneous descriptions are similar to Turing's *complete configurations*, and *transitions* '→' represent Turing's *moves* (14, last line of page 232). The keypoint here is that the last instantaneous description α_p must be *terminal*, i.e., no transition is possible from it (4, Definition 1.9). In other words, "*the machine interprets the absence of an instruction as a stop order*" when reaching α_p (4, footnote 1 in page 7). This is the usual notion of computation with TMs today.

▶ Remark 1 (Church's λ-calculus). The idea of a computation that halts was proposed in 1936 by Church (1) for his λ-calculus. Computations in the λ-calculus (*effective calculations* in Church's terminology (1)) start with a given expression, perform some *conversion* (nowadays we would rather say β-*reduction*, or just *reduction*) steps, and finish when a *normal form* (i.e., an expression that cannot be further reduced) is obtained. Thus, Church's notion of computation assumes that only *finite* sequences of reductions are considered.

▶ Remark 2 (Kleene's description of the operation of a TM). In (7) Turing's complete configurations are called *situations* by Kleene. TMs are used as follows (7, page 358):

> *The change from the initial situation to the terminal situation (when there is one) may be called the* operation *performed by the machine.*

Here, the *terminal situation* or *output* is one "*in which [the machine] stops*". Changes between intermediate situations are performed in the usual way, using Turing's *moves*. Hence, Davis' "computation of a TM" is analogous to Kleene's "operation performed by a TM".

4 Decision problems about Turing machines

In his paper (14), Turing enunciates two problems about his machines and proves them undecidable. The first problem considered by Turing is given on page 247, as follows:

> *Is there a machine $\mathcal{D}$ which, when supplied with the description of any computing machine $\mathcal{M}$ will test this description and if $\mathcal{M}$ is circular will mark the description with the symbol 'u' and if it is circle-free will mark it with 's'?*

Here, 'u' and 's' represent an *unsatisfactory* or *satisfactory* verdict of $\mathcal{D}$ about $\mathcal{M}$ being circle-free. Copeland coined this as the *Satisfactoriness Problem* (2, page 36). In the following, we refer to it as SATIS. The second problem posed by Turing is as follows (14, page 248):

> *there can be no machine $\mathcal{E}$ which, when supplied with the description of an arbitrary machine $\mathcal{M}$, will determine whether $\mathcal{M}$ ever prints a given symbol (0 say).*

Davis uses *printing problem* (PRINT in the following) to refer to it (4, page 70). In the following, for each machine $\mathcal{M}$, we say that $\mathsf{Print}(\mathcal{M})$ is *true* iff $\mathcal{M}$ prints a given symbol (e.g., 0) during its (finite or infinite) execution (and often say that $\mathcal{M}$ is a *printing* machine).

4.1 The halting problem

According to Davis' formulation, the halting problem for a TM $\mathcal{M}$ aims (4, page 70)

> *to determine whether or not $\mathcal{M}$, if placed in a given initial state, will eventually halt.*

This is the usual understanding of the halting problem for TMs (HALT in the following). Davis proves it undecidable in Chapter 5, Theorem 2.2. In the following, we say that $\mathsf{Halt}(\mathcal{M})$ is *true* iff $\mathcal{M}$ halts when placed in a given initial state (and say that $\mathcal{M}$ is a *halting* machine).

4.2 Relationship between *printing* and *halting* problems

There are printing machines $\mathcal{M}$ which do not halt. For instance, $\mathcal{P}_{C3}$ and $\mathcal{P}_{CF}$. Vice versa, some halting machines print nothing (e.g., $\mathcal{P}_{C1}$). Thus, *printing and halting machines do* not *coincide*. Moreover, *there are Turing Machines for which the printing problem is decidable, but the halting problem is not* (8, Section 4.4.2). Thus, printing and halting problems address different issues and exhibit important conceptual and technical differences.

There are, however, related according to their *undecidability degree*. In his PhD thesis, see (15), Turing introduced the idea of comparing different unsolvable problems by means of a variant of his 1936 *a-machines* (called *o-machines*) where undecidable questions could be 'answered' with the help of an *oracle*, the main idea being that questions Q which are undecidable by a TM could be answered by an *extended* version with some additional knowledge provided by the oracle. Such questions would be strictly harder than problems P which are known to be solvable by a TM. Post coined the term *degree of unsolvability* (10, page 289), although the problem had also been investigated by Kleene (see, e.g., (7) and the references therein). PRINT and HALT have the same *undecidability degree* (the *least* one, denoted $\mathbf{0}'$, being $\mathbf{0}$ reserved for *decidable* problems). However, the undecidability degree of SATIS is strictly bigger: $\mathbf{0}''$. This was conjectured by Post in (11). We prove it, to the best of our knowledge for the first time, in (8, Section 5.2). The full hierachy of the considered problems regarding their undecidability degrees is

$$\mathsf{HALT} \equiv \mathsf{PRINT} \equiv \mathbf{0}' < \mathbf{0}'' \equiv \mathsf{UHALT} \equiv \mathsf{SATIS}$$

where UHALT is the *uniform halting problem*, i.e., the problem investigating whether a TM halts for *every* input, see, e.g., (9, page 57).

5 Church and Kleene: around the halting problem

Church proved that "*the property of a well-formed formula [i.e., a λ-expression], that it has a normal form is not recursive*" (1, Theorem XVIII). In an appendix to (14), Turing proves that "every λ-definable sequence is computable" and vice versa. A sequence γ of digits in $\{0,1\}$ is *λ-definable* if there is a λ-expression M_γ such that for all $n \in \mathbb{N}$, with M_n the usual encoding of natural numbers n as λ-expressions (1, page 347), $(M_\gamma\ M_n)$ reduces to the (λ-expression representing the) n-th digit of γ. Then, Turing "*constructs a machine $\mathcal{L}$ which, when supplied with the formula M_γ writes down the sequence γ*". Such a machine works in two steps (14, pages 263-264): for each $n \in \mathbb{N}$, (i) the expression $(M_\lambda\ M_n)$ is built and then (ii) a submachine $\mathcal{L}_2$ is applied to obtain successively all expressions into which $(M_\lambda\ M_n)$ is convertible. Each obtained expression is checked to decide whether (ii.1) a normal form has been obtained, and then $\mathcal{L}$ prints 1 or 0, respectively, thus moving to the $n+1$ component of the sequence, or (ii.2) the conversion (i.e., reduction) process issued by $\mathcal{L}_2$ should continue instead. By hypothesis, $(M_\gamma\ M_n)$ is convertible to the representation of the n-digit, 0 or 1, of γ (i.e., $(M_\gamma\ M_n)$ is normalizing); thus, the move of $\mathcal{L}$ to produce the $n+1$-th component of γ is guaranteed. Note that $\mathcal{L}$ does *not* halt, as sequences γ are (by definition) infinite.

Church's result above is relevant for analyzing the behavior of a variant $\mathcal{L}_2'$ of $\mathcal{L}_2$ which, when fed with a (description of a) λ-expression performs a (leftmost) outermost conversion step and then, repeatedly, (i) halts if a normal form is obtained, or else (ii) performs a new outermost step. The halting problem for $\mathcal{L}_2'$ is undecidable, as $\mathcal{L}_2'$ halts when applied on the description of a λ-expression if and only if it has a normal form, which is undecidable (implementing outermost steps in $\mathcal{L}_2'$ is essential for the *if* part, as outermost reduction is normalizing (6, Section 3D)). However, although the undecidability of the halting problem

for TMs follows from Church's result, this is not mentioned in Turing's appendix. Again, this suggests that Turing's focus was other. Also, in his review of Turing's paper (*Journal of Symbolic Logic* 2(1):42–43, 1937) Church did not draw any connection between normalization of λ-expressions and halting TMs.

▶ Remark 3 (Kleene's statement). In his 1952 book, Kleene makes the following statement (7, Chapter XIII, Section 71)

> there is no algorithm for deciding whether any given machine, when started from any given situation, eventually stops.

which corresponds to what we currently know as the *halting problem*, although the word *stops* is used instead of *halts*. Kleene justifies this statement by using Turing's arguments, but without citing any of his results. This suggests that Kleene does *not* identify any of Turing's results in (14) as equivalent to his statement.

6 Bibliographical analysis

The (necessarily incomplete) analysis of the computability and undecidability literature between 1936 and 1958 shows that, apparently, the term "halting problem" was not used before the publication of (4). A number of journals and repositories in the fields of mathematics, logic, and computer science with facilities to search for (prefixes of) words in their archives were considered and '*halt*' tried on the corresponding search devices for the period 1936-1958. The obtained outcomes from these journals and repositories had nothing to do with the halting problem. Thus, *apparently, the term "halting problem" was not used in the literature before the publication of (4) (with one exception; see the discussion below).*

An early reference by Davis to a preliminary ("in preparation") version of (4) can be found in (3). No mention of the halting problem is made, though. Remarkably, Rogers' 1957 book (12) includes what apparently is the *first printed reference* to the halting problem. He writes (12, page 19):

> *There is no effective procedure by which we can tell whether or not a given effective computation will eventually come to a stop. (Turing refers to this as the unsolvability of the* halting problem *for machines. This and the existence of the universal machine are the principal results of Turing's first paper.)*

Clearly, Rogers does not claim the authorship of the notion, as he refers to Turing instead. But Copeland already mentioned that Turing never used the word "halting" in (14). In the preface Rogers says that the manuscript of (4) was available to him when preparing the book (12, page 3). One year later, in (13, page 333), with (4) already published, he mentions the halting problem again but he does not mention Turing anymore; instead he cites (4).

7 Conclusions

The subject of Turing's paper was "*ostensibly the computable numbers*" (14, page 230). He defines a number as computable if "*its decimal part can be written down by a machine*"; and points to numbers like π (with infinitely many decimals without any repetition pattern), as computable (14, page 230). Thus, if Turing's focus was in infinitary computations, it is not surprising that investigating halting machines was not a priority. Furthermore, Turing needed nonterminating machines but not *any* nonterminating machine: only 'productive' ones in the sense that infinitely many figures are printed during the computation. He defines a number

as "*computable if it differs by an integer from the number computed by a circle-free machine*". This is the focus of his first undecidability result: SATIS tries to determine whether a given machine is circle-free (satisfactory!), and hence able to generate a sequence corresponding to a computable number. Halting machines were just *unsatisfactory*, see also (5, Section 2.4.2).

Summarizing: Church included termination as part of his notion of effective calculation (1), but this concerns λ-calculus as computational mechanism. When introducing his machines (14), Turing was not interested in halting machines and his notion of computation focused on the generation of infinite sequences of figures instead. His undecidability results were according to this. Church had already proved termination of effective calculations undecidable. Although this could be used to prove the halting problem of TMs undecidable, no attempt to do it was made either by Turing or Church. Kleene considered *coming to a stop* as an ingredient of the notion of computation (*operation* in Kleene's terminology) with TMs. In his 1952 book, he formulated and informally proved a statement (Remark 3) which we easily recognize as what we call halting problem now (but 'stops' is used instead of 'halts'). Davis' notion of computation also required a halting behavior of the machines. He formulated the halting problem in its current wording (thus coining the term we currently use), and proved it undecidable. However, except for the word *halt* the result was in Kleene's 1952 book (7).

References

1 Alonzo Church. An unsolvable problem of elementary number theory. *The American Journal of Mathematic*, 58:345–36, 1936.

2 Jack B. Copeland. *The essential Turing: Seminal Writings in Computing, Logic, Philosophy, Artifical Intelligence and Artificial Life: Plus The Secrets of Enigma*. Oxford University Press, 2004.

3 Martin D. Davis. *A note on universal Turing machines*, pages 167–175. Princeton University Press, 1956.

4 Martin D. Davis. *Computability and Unsolvability*. McGraw-Hill, 1958.

5 Liesbeth De Mol. Turing Machines. In Edward N. Zalta, editor, *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, Winter 2019 edition, 2019.

6 J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda-Calculus*. Cambridge University Press, 1986.

7 Stephen C. Kleene. *Introduction to Metamathematics*. Wolters-Noordof and North-Holland, 1952.

8 Salvador Lucas. The origins of the halting problem. *Journal of Logical and Algebraic Methods in Programming*, 121:100687, 2021.

9 Zohar Manna. *Mathematical Theory of Computation*. McGraw-Hill (reprinted by Dover, 2003), 1974.

10 Emil Post. Recursively enumerable sets of positive integers and their decision problems. *Bulletin of the American Mathematical Society*, 50:284–316, 1944.

11 Emil Post. Recursive Unsolvability of a Problem of Thue. *Journal of Symbolic Logic*, 12:1–11, 1947.

12 Hartley Rogers. *Theory of recursive functions and effective computability. Vol. 1*. Technology Store, Cambridge, MA, 1957.

13 Hartley Rogers. Gödel numberings of partial recursive functions. *Journal of Symbolic Logic*, 33:331–341, 1958.

14 Alan M. Turing. On computable numbers, with an application to the Entscheindungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1936.

15 Alan M. Turing. Systems of logic based on ordinals. *Proceedings of the London Mathematical Society*, 45:161–228, 1939.

de Vrijer's Measure for SN of $\lambda^{\rightarrow}$ in Scheme

Nachum Dershowitz ✉
Tel Aviv University, Israel

Abstract

We contribute a Scheme program for de Vrijer's proof of strong normalization of the simply-typed lambda calculus.

2012 ACM Subject Classification Theory of computation → Logic; Theory of computation → Type theory

Keywords and phrases Strong normalization, simply typed lambda calculus

Acknowledgements I thank the members of the lambda-calculus and rewriting group at Vrije University, Amsterdam, for the enjoyable meetings we had together shortly before the pandemic struck.

> Mr Howard, it shouldn't be too difficult to find a right ordinal assignment showing SN.
>
> Kurt Gödel to Bill Howard, as reported to me by Henk Barendregt

1 Gödel's Koan

Gödel's "koan" [4, 3, Problem 19], put forward by Jean-Jacques Lévy in 1991 and by others before and after (see [6, Problem 26]), asks for an easy, intuitive way to assign ordinals—be they natural numbers or transfinite ordinals—to terms of the simply-typed lambda calculus such that each (β-) reduction step of a term yields a smaller ordinal. There are a fair number of proofs of the strong normalization (SN), a.k.a. uniform termination, of (well-) typed lambda terms, including [9, 11, 5, 8, 7, 10], but none yet that meet the desideratum of intuitiveness. All the same, one compelling assignment that proves termination is that designed by Roel de Vrijer [2]. It measures a term by an overestimate of the maximum number of reduction steps, at the same time constructing a function for each term to provide that value when said term is applied to another.

I found it nontrivial to program de Vrijer's measure. Hence, this note. I follow a summary formulation by Henk Barendregt [1], and I program in Scheme (a variant of Lisp) for its built-in evaluation of lambda expressions. A lazy language might have been easier to program in.

2 de Vrijer's Measure

de Vrijer's measure $[\![e]\!]$ of typed term e generally consists of two components, what we call the "dot part" or "dot measure", $[\![e]\!]^\bullet$, and the "star part" or "star measure", $[\![e]\!]^*$. The star measure is an integer, an upper bound on the maximum length of a reduction sequence starting from e. But when calculating $[\![e]\!]$ recursively, one also needs to know what happens when e is a subterm embedded in a more complicated expression. The dot part serves this purpose. It is a function that knows how to compute the two-part measure $[\![et]\!]$ for any application of the current expression e to another expression t whose measure $[\![t]\!]$ is given. In other words, $[\![et]\!] = [\![e]\!]^\bullet([\![t]\!])$. The central consideration is that $mn + n + 1$ is an upper bound on the length of a reduction of a composition $(\lambda x.M)N$, where n bounds reductions

of N and x appears at most m times in a reduct of M. So, each of the m copies of N can be reduced n times, plus one more step for the application of M to N, which first substitutes N for each (free) occurrence of x in the reduct. The additional term n in the (potentially over-) estimate is to cover the case when $m = 0$ but N is reduced nonetheless. The multiplier m is not computed in advance; rather, n gets plugged into M and tallied as its measure gets expanded. All this needs to be computed within an environment that tracks variable bindings dictated by the larger, enclosing term while building the measure recursively.

The whole measure can also be just a number, rather than a pair. In particular, the measure $[\![x]\!]$ of an unbound variable x is 0.

As a minimalist example, $[\![\lambda x^o.x]\!]$ (o is the base type) is essentially $\langle \lambda n.n+1, 0\rangle$. Since $\lambda x^o.x$ is in normal form, the star part $[\![\lambda x^o.x]\!]^*$ is 0. But if it's applied to a term with maximum reduction-sequence length n, then the full reduction will have an additional β step, so $[\![\lambda x^o.x]\!]^\bullet = \lambda n.n+1$.

Of course, one needs to prove that the measure in fact decreases with each β-step, regardless of the assignments (of measures) to free variables in the measure, for which see [2, Reduction Lemma, §3.5] or [1, Prop. 3.7].

de Vrijer [2, §4] also provides a more complicated, but precise, measure, which we do not address here.

The code is presented in the next section, followed by an example. Code and examples are made available at `http://nachum.org/SN.scm`.

3 My Scheme Code

3.1 Basics

A lambda term can be

(1) a typed variable, given as a variable symbol paired with a type expression, `(: x t)`,

(2) an application of one term to another, `(f t)`, or

(3) an abstraction (definition), `(lambda v t)`, comprising the atom λ, a bound typed variable (parameter) `v`, and a body `t`, which is itself a lambda term.

Types are either basic `o` or triples `(-> a b)`. Assignments are (association) lists of (dotted) pairs `(v . e)`, each giving a variable binding `v` $\mapsto$ `e`.

Accordingly, the basic constructors, destructors, and tests have the following simple definitions:

```
(define (atom? x) (not (list? x))) ; is-atom?
(define (var? x)
  (or (atom? x) (and (list? x) (equal? (car x) ':)))) ; is-variable?
(define (abs? x) (and (list? x) (equal? (car x) 'lambda))) ; is-abstraction?
(define (comp x y) (list x y)) ; application as list of two expressions
(define (fun x) (first x)) ; function part of application
(define (arg x) (second x)) ; argument part of application
(define (var x) (second x)) ; name of typed variable
(define (param x) (second (var x))) ; variable of abstraction
(define (kind x) (third (var x))) ; type of parameter
(define (body x) (third x)) ; body of abstraction
(define (mapsto x y) (cons x y)) ; binding as pair
(define (from y) (second y)) ; variable of binding
(define (to y) (third y)) ; assignment to variable
```

The following function extracts, or constructs, the type of a lambda expression, depending on its form:

```
(define (type x)
  (cond
    ((var? x) (third x))
    ((abs? x) `(-> ,(kind x) ,(type (body x))))
    (#t (third (type (fun x))))))
```

where ` is quote and , is unquote (evaluate first, before quoting). The case statement `cond` comprises a list of conditions plus values to return, and `#t` denotes the truth value, true.

We will use the form `(@ f t)`, which when executed forces the evaluation of `f` and `t` followed by application of the former to the latter:

```
(define (@ f t) ((eval f) (eval t))) ; apply!
```

where `eval` is a Scheme primitive.

3.2 The Two-Part Measure

The measure has two parts and will be represented by the expression `(L d s)`, where `d` is its dot part and `s`, the star part. Its constructors and destructors are

```
(define (L x y) `(L ,x ,y))
(define (dot x) (if (atom? x) x (second x))) ; dot part of measure
(define (star x) (if (atom? x) x (third x))) ; star part of measure
```

When the measure is a number, not a pair, both its parts are just that.

Given an expression `x` in an environment with assignments `v`, the following computes the measure by induction on the form of the expression:

```
(define (bra x v) ; compute measure
  (cond
    ((var? x) (value v x))
    ((abs? x)
     (let ((f (gensym)))
       (L `(lambda (,f)
             (add ,(bra (body x)
                        (cons (mapsto (var x) f) v))
                  (+ (star ,f) 1)))
          `(star ,(bra (body x)
                       (cons (mapsto (var x) (c (type (var x)) 0)) v))))))
    (#t `(@ (dot ,(bra (fun x) v)) ,(bra (arg x) v)))))
```

This is the heart of the method.

(1) For variables, one looks up what may already have been assigned to it, or else one creates a new initial valuation `c` that depends on the variable's type, as will be seen below.

(2) For abstractions, one constructs dot and star components. For its dot part, a new assignment is attached to the environment, the measure of the body is determined, and 1 is added for one beta-reduction step when it's applied, plus the bound on the steps needed to bring the function itself to normal form. The `gensym` command creates a new formal parameter for the dot measure, which is assigned in the environment to

the variable of the abstraction that will also occur in the measure. (See below for the mechanism of addition.) Its star component is the star of the body with a promise for the assignment to the parameter.

(3) The last case is application of a function term to an argument term. This is exactly where the dot measure comes into play. The dot part of the pre-computed function's measure is applied to the argument's measure—whose evaluation has been delayed until now—to obtain the full measure for the combined application term.

The above `bra` function makes use of the following to create a new measure that increases both parts of the given measure `f` by a given integral amount `n`:

```
(define (add f n) ; add natural n to measure f
  (if (number? f)
      (+ f n)
      (let ((g (gensym)))
        (L `(lambda (,g) (add (@ (dot ,f) ,g) ,n))
           (+ (star f) n)))))
```

The star measure just gets added to, but the dot measure requires a new function that will do the addition when the time comes.

Applying an assignment is easy, treating the assignment as lookup in the association list of stored bindings:

```
(define (value v x) ; apply valuation v to variable x
  (if (assoc x v)
      (cdr (assoc x v))
      (c (type x) 0))) ; initial valuation
```

When necessary, viz. when the variable is not listed, this creates a new (typed) valuation `c`, using the following:

```
(define (c y n)
  (if (equal? y 'o) ; base type
      n
      (let ((f (gensym)))
        (L `(lambda (,f) (c (quote ,(to y)) (+ ,n (star ,f))))
            n))))
```

For a variable x of non-base type $\sigma \to \tau$, this valuation considers what happens when an instance of x is applied to some term f of type σ. The dot measure of x should return a valuation for (possibly compound) type τ that also incorporates the length of reductions given by the star measure $[\![f]\!]^*$.

Finally, to measure a top-level expression `x`, first construct and evaluate its measure in a pristine (empty) environment and then take its star part:

```
(define (o x) (star (eval (bra x '()))))) ; the measure
```

4 Barendregt's Examples

I get the same values for all of Henk's examples [1] as he obtained manually, except for $(\lambda f^{o\to o}.\lambda x^{o}.fx)(\lambda x^{o\to o}.x)$, for which the above program calculates 2 as its star measure rather than 1, as therein. In our Scheme formalization this term is

```
(comp `(lambda ,f1 (lambda ,x0 ,(comp f1 x0))) `(lambda ,x1 ,x1))
```

The full, computed measure for this, just prior to a final evaluation-cum-application step `@`, looks like the following:

```
(@ (dot (L (lambda (g75)
             (add (L (lambda (g76)
                       (add (@ (dot g75) g76)
                            (+ (star g76) 1)))
                     (star (@ (dot g75) 0)))
                  (+ (star g75) 1)))
           (star (L (lambda (g78)
                      (add (@ (dot (L (lambda (g77)
                                        (c 'o (+ 0 (star g77))))
                                   0))
                               g78)
                           (+ (star g78) 1)))
                 (star (@ (dot (L (lambda (g77)
                                    (c 'o (+ 0 (star g77))))
                                  0))
                          0))))))
   (L (lambda (g79) (add g79 (+ (star g79) 1)))
      (star (L (lambda (g80) (c 'o (+ 0 (star g80))))
               0))))
```

Simplifying, this is equivalent to

```
(@ (lambda (g75)
     (add (L (lambda (g76)
               (add (@ (dot g75) g76)
                    (+ (star g76) 1)))
             (star (@ (dot g75) 0)))
          (+ (star g75) 1)))
   (L (lambda (g79) (add g79 (+ (star g79) 1)))
      0))
```

Evaluating and simplifying further yields the measure

```
(L (lambda (g76) (add (add (add g76 (+ (star g76) 1)) (+ (star g76) 1)) 1))
   2)
```

The dot part of this—when applied to a measure that is a plain number—is tantamount to $\lambda n.3n + 3$. Its star component is 2. Indeed, evaluating the expression takes two steps: $(\lambda f.\lambda x.fx)(\lambda x.x) \rightarrow_\beta \lambda x.(\lambda x.x)x \rightarrow_\beta \lambda x.x$. As Henk says [1]: "It isn't completely trivial to compute these: easy to make mistakes."

> The truth is you don't really understand something until you've taught it to a computer, until you've been able to program it.
>
> Don Knuth (2008)

References

1 Henk Barendregt. Digesting the proof of Roel de Vrijer that $\lambda_{\rightarrow} \models \mathrm{SN}$. Unpublished note, April 2019. URL: http://nachum.org/Henk.pdf.

2 Roel C. de Vrijer. Exactly estimating functionals and strong normalization. *Indagationes Mathematicae*, 49:479–493, 1987. URL: https://core.ac.uk/reader/82154640.

3 Nachum Dershowitz. The RTA list of open problems, 2009–2021. URL: https://www.cs.tau.ac.il/~nachum/rtaloop/.

4 Nachum Dershowitz, Jean-Pierre Jouannaud, and Jan Willem Klop. Open problems in rewriting. In R. Book, editor, *Proceedings of the Fourth International Conference on Rewriting Techniques and Applications (Como, Italy)*, volume 488 of *Lecture Notes in Computer Science*, pages 445–456, Berlin, April 1991. Springer-Verlag. URL: https://www.researchgate.net/publication/2441091.

5 Robin O. Gandy. Proofs of strong normalization. In *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 457–477. Academic Press Limited, 1980.

6 Ryu Hasegawa, Luca Paolini, and Paweł Urzyczyn. TLCA list of open problems, July 2014. URL: http://tlca.di.unito.it/opltlca/.

7 Assaf J. Kfoury and Joe B. Wells. New notions of reduction and non-semantic proofs of strong β-normalization in typed λ-calculi. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science*, pages 311–321, 1995. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.37.9949&rep=rep1&type=pdf.

8 Jan Willem Klop. *Combinatory Reduction Systems*, volume 127 of *Mathematical Centre Tracts*. Mathematisch Centrum, Amsterdam, 1980.

9 Luis Elpidio Sanchis. Functionals defined by recursion. *Notre Dame Journal of Formal Logic*, VIII(3):161–174, July 1967. doi:10.1305/ndjfl/1093956080.

10 Morten Heine Sørensen. Strong normalization from weak normalization in typed λ-calculi. *Information and Computation*, 133:35–71, 1997. doi:10.1006/inco.1996.2622.

11 William W. Tait. A realizability interpretation of the theory of species. In Rohit Parikh, editor, *Logic Colloquium*, volume 453 of *Lecture Notes in Mathematics*, pages 240–251. Springer, Boston, 1975.

Polynomial Loops: Termination and Beyond

Florian Frohn
Max Planck Institute for Informatics, Saarland Informatics Campus, and AbsInt GmbH, Saarbrücken, Germany

Jürgen Giesl
LuFG Informatik 2, RWTH Aachen University, Aachen, Germany

Marcel Hark
LuFG Informatik 2, RWTH Aachen University, Aachen, Germany

Abstract

We consider triangular weakly non-linear loops (*twn*-loops) over subrings $\mathcal{S}$ of $\mathbb{R}_{\mathbb{A}}$, where $\mathbb{R}_{\mathbb{A}}$ is the set of all real algebraic numbers. Essentially, the body of such a loop is a single assignment $\begin{bmatrix} x_1 \\ \cdots \\ x_d \end{bmatrix} \leftarrow \begin{bmatrix} c_1 \cdot x_1 + pol_1 \\ \cdots \\ c_d \cdot x_d + pol_d \end{bmatrix}$ where each x_i is a variable, $c_i \in \mathcal{S}$, and each pol_i is a (possibly non-linear) polynomial over $\mathcal{S}$ and the variables $x_{i+1}, \ldots, x_d$. We present a reduction from the question of termination on all inputs to the existential fragment of the first-order theory of $\mathcal{S}$ and $\mathbb{R}_{\mathbb{A}}$. For loops over $\mathbb{R}_{\mathbb{A}}$, our reduction entails decidability of termination. For loops over $\mathbb{Z}$ and $\mathbb{Q}$, it proves semi-decidability of non-termination.

Furthermore, we show that the *halting problem*, i.e., termination on a given input, is decidable for *twn*-loops over any subring of $\mathbb{R}_{\mathbb{A}}$. This also allows us to compute *witnesses for non-termination*.

Moreover, we present the first computability results on the *runtime complexity* of such loops. More precisely, we show that for *twn*-loops over $\mathbb{Z}$ one can always compute a polynomial f such that the length of all terminating runs is bounded by $f(\|(x_1, \ldots, x_d)\|)$, where $\|\cdot\|$ denotes the 1-norm. This result implies that the runtime complexity of a terminating triangular *linear* loop over $\mathbb{Z}$ is at most linear.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Problems, reductions and completeness

Keywords and phrases Polynomial Loops, Closed Forms, Decidability of Termination, Halting Problem, Complexity Analysis,

Funding funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 389792660 as part of TRR 248, by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 235950644 (Project GI 274/6-2), and by the DFG Research Training Group 2236 UnRAVeL

Acknowledgements We thank Alberto Fiori and Arno van den Essen for useful discussions.

1 Introduction

We consider loops of the form

$$\textbf{while } \varphi \textbf{ do } \vec{x} \leftarrow \vec{u}. \tag{1}$$

Here, $\vec{x}$ is a vector of pairwise different variables $x_1, \ldots, x_d$ that range over a ring $\mathbb{Z} \leq \mathcal{S} \leq \mathbb{R}_{\mathbb{A}}$, where $\leq$ denotes the subring relation and $\mathbb{R}_{\mathbb{A}}$ is the set of real algebraic numbers. Moreover, $\vec{u} \in (\mathcal{S}[\vec{x}])^d$, i.e., $\vec{u}$ is a vector of polynomials over $\vec{x}$ with coefficients in $\mathcal{S}$. The condition φ is a propositional formula over the atoms $\{pol \triangleright 0 \mid pol \in \mathcal{S}[\vec{x}], \triangleright \in \{\geq, >\}\}$. The motivation for the restriction to such "single-path" loops is that termination is undecidable for "multi-path" loops which contain conditionals [1].

We often represent a loop (1) by the tuple $(\varphi, \vec{u})$ of the *condition* φ and the *update* $\vec{u} = (u_1, \ldots, u_d)$. In this paper, $(\varphi, \vec{u})$ is always a loop on $\mathcal{S}^d$ using the variables $\vec{x} = (x_1, \ldots, x_d)$ where $\mathbb{Z} \leq \mathcal{S} \leq \mathbb{R}_{\mathbb{A}}$. For any entity s and terms $\vec{t}$, we often abbreviate $s[\vec{x}/\vec{t}]$ by $s(\vec{t})$.

Previous works on decidability of termination were restricted to conditions only containing conjunctions or defining compact sets (see [2] for a discussion of related work). Moreover, if considering non-linear loops, those works only apply to loops over the reals. In this paper, we regard linear and non-linear loops whose loop conditions can be arbitrary propositional formulas over polynomial inequations, i.e., they may also contain disjunctions and define non-compact sets. Furthermore, we study the decidability of termination for non-linear loops over $\mathbb{Z}$, $\mathbb{Q}$, and $\mathbb{R}_{\mathbb{A}}$. In this way, we identify new sub-classes of loops of the form (1) where (non-)termination is (semi-)decidable (see Sect. 3).

Still, non-termination proofs should be accompanied by a *witness of non-termination*, i.e., a non-terminating input that can, e.g., be used for debugging. However, most existing (semi-)decision procedures for (non-)termination do not yield such witnesses. To close this gap, we prove the novel result that the *halting problem* for *twn*-loops is decidable (see Sect. 4), i.e., we show how to decide whether a loop terminates on a *fixed* input. Thus, we obtain a technique to enumerate all witnesses for non-termination of a loop recursively.

For terminating inputs, the question *how fast* the loop terminates is of high interest. Thus, many automated techniques to derive bounds on the *runtime* of programs have been proposed but these are usually incomplete. In contrast, we present a *complete* technique to derive polynomial upper bounds on the runtime of *twn*-loops over $\mathbb{Z}$ (see Sect. 5). As a corollary we obtain that triangular *linear* loops have at most linear runtime. For the full papers on our results, we refer to [2, 3].

2 Preliminaries

Let $(\varphi, \vec{u})$ be a loop over $\mathcal{S}$ and $\vec{e} \in \mathcal{S}^d$. If $\forall n \in \mathbb{N}.\ \varphi(\vec{u}^n(\vec{e}))$ holds,[1] then $\vec{e} \in \mathcal{S}^d$ is a *witness for non-termination*. Otherwise, the loop terminates on $\vec{e}$. If $(\varphi, \vec{u})$ does not have any witnesses for non-termination, then $(\varphi, \vec{u})$ *terminates*.

Given an assignment $\vec{x} \leftarrow \vec{u}$, the relation $\succ_{\vec{u}} \in \mathcal{V}(\vec{u}) \times \mathcal{V}(\vec{u})$ is the transitive closure of $\{(x_i, x_j) \mid i, j \in \{1, \ldots, d\}, i \neq j, x_j \in \mathcal{V}(u_i)\}$, i.e., $x_i \succ_{\vec{u}} x_j$ means that x_i depends on x_j. A loop $(\varphi, \vec{u})$ is *triangular* if $\succ_{\vec{u}}$ is well founded. So the restriction to triangular loops prohibits "cyclic dependencies" of variables (e.g., where the new values of x_1 and x_2 both depend on the old values of x_1 and x_2). Furthermore, $(\varphi, \vec{u})$ is *weakly non-linear* if there is no $1 \leq i \leq d$ such that x_i occurs in a non-linear monomial of u_i.

A *twn*-loop is triangular and weakly non-linear. So in other words, by permuting variables every *twn*-loop can be transformed to the form $\begin{bmatrix} x_1 \\ \cdots \\ x_d \end{bmatrix} \leftarrow \begin{bmatrix} c_1 \cdot x_1 + pol_1 \\ \cdots \\ c_d \cdot x_d + pol_d \end{bmatrix}$ where $c_i \in \mathcal{S}$ and $pol_i \in \mathcal{S}[x_{i+1}, \ldots, x_d]$. If $(\varphi, \vec{u})$ is weakly non-linear and each c_i is non-negative, then $(\varphi, \vec{u})$ is called *non-negative*. A *tnn*-loop is triangular and non-negative (and thus, also weakly non-linear). Triangularity and non-negativity ensure that we can compute closed forms for the n-fold iterated update, i.e., for any *tnn*-loop $(\varphi, \vec{u})$ we can compute an expression $\vec{q}$ in the variables $\vec{x}$ and a distinguished variable n such that $\vec{u}^m(\vec{e}) = \vec{q}[n/m, \vec{x}/\vec{e}]$ for all $m \in \mathbb{N}$ and all $\vec{e} \in \mathcal{S}^d$.

When analyzing the termination behavior of *twn*-loops, it is enough to only consider *tnn*-loops. The reason is that for a *twn*-loop $(\varphi, \vec{u})$ the *chained* loop $(\varphi_{\text{ch}}, \vec{u}_{\text{ch}}) = (\varphi \wedge \varphi(\vec{u}), \vec{u}(\vec{u}))$

[1] Here, $\vec{u}^0(\vec{e}) = \vec{e}$ and $\vec{u}^{n+1}(\vec{e}) = \vec{u}^n(\vec{u}(\vec{e}))$.

is *tnn*. Moreover, $(\varphi, \vec{u})$ terminates on $\vec{e}$ iff $(\varphi_{\text{ch}}, \vec{u}_{\text{ch}})$ terminates on $\vec{e}$ (see [2] for details).

When computing closed forms for the n-fold update of *tnn*-loops, one obtains so-called *poly-exponential expressions*. These expressions are sums of terms of the form $\alpha \cdot n^a \cdot b^n$ with $a \in \mathbb{N}$, $b \in \mathcal{S}_{>0}$, and $\alpha \in Q_{\mathcal{S}}[\vec{x}]$ where $Q_{\mathcal{S}}$ is the quotient field of $\mathcal{S}$.

▶ **Example 1.** Consider the loop

$$\textbf{while } x_1 + x_2^2 > 0 \textbf{ do } \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \leftarrow \begin{bmatrix} x_1 + x_2^2 \cdot x_3 \\ x_2 - 2 \cdot x_3^2 \\ x_3 \end{bmatrix}. \tag{2}$$

This loop is *tnn* as $\succ_{(2)} = \{(x_1, x_2), (x_1, x_3), (x_2, x_3)\}$ is well founded. Moreover, every variable x_i occurs linearly with a non-negative coefficient in its corresponding update.

A closed form for the update after $n \in \mathbb{N}$ loop iterations is:

$$\vec{q} = \begin{bmatrix} \frac{4}{3} \cdot x_3^5 \cdot n^3 + \left(-2 \cdot x_3^5 - 2 \cdot x_2 \cdot x_3^3\right) \cdot n^2 + \left(x_2^2 \cdot x_3 + \frac{2}{3} \cdot x_3^5 + 2 \cdot x_2 \cdot x_3^3\right) \cdot n + x_1 \\ -2 \cdot x_3^2 \cdot n + x_2 \\ x_3 \end{bmatrix}$$

3 Reducing Termination of *tnn*-Loops to $\text{Th}_\exists(\mathcal{S}, \mathbb{R}_\mathbb{A})$

In the following, let $(\varphi, \vec{u})$ be a *tnn*-loop and let $\vec{q}$ be the closed form of $\vec{u}^n$. We now show how to encode termination of $(\varphi, \vec{u})$ on $\mathcal{S}$ into a $\text{Th}_\exists(\mathcal{S}, \mathbb{R}_\mathbb{A})$-formula. Here, $\text{Th}_\exists(\mathcal{S}, \mathbb{R}_\mathbb{A})$ is the *existential fragment of the first-order theory of* $\mathcal{S}$ *and* $\mathbb{R}_\mathbb{A}$, i.e., the set of all formulas $\exists \vec{y}' \in \mathbb{R}_\mathbb{A}^{k'}, \vec{y} \in \mathcal{S}^k. \psi$, with a propositional formula ψ over $\{p \triangleright 0 \mid p \in \mathbb{Q}[\vec{y}', \vec{y}, \vec{z}], \triangleright \in \{\geq, >\}\}$ where $k', k \in \mathbb{N}$ and the variables $\vec{y}'$, $\vec{y}$, and $\vec{z}$ are pairwise disjoint.

We use the concept of *eventual non-termination*, where the loop condition may be violated finitely often, which is clearly equivalent to non-termination [5]. Expressed using the closed form $\vec{q}$ containing the variables $\vec{x}$ and n, $(\varphi, \vec{u})$ is eventually non-terminating on $\mathcal{S}$ iff $\exists \vec{x} \in \mathcal{S}^d$, $n_0 \in \mathbb{N}$. $\forall n \in \mathbb{N}_{>n_0}$. $\varphi(\vec{q})$.

▶ **Example 2.** Continuing Ex. 1, (2) is eventually non-terminating iff

$$\exists x_1, x_2, x_3 \in \mathcal{S},\ n_0 \in \mathbb{N}.\ \forall n \in \mathbb{N}_{>n_0}.\ pe > 0, \tag{3}$$

where $pe = \left(\frac{4}{3} \cdot x_3^5\right) \cdot n^3 + \left(-2 \cdot x_3^5 - 2 \cdot x_2 \cdot x_3^3 + 4 \cdot x_3^4\right) \cdot n^2 + \left(x_2^2 \cdot x_3 + \frac{2}{3} \cdot x_3^5 + 2 \cdot x_2 \cdot x_3^3 - 4 \cdot x_2 \cdot x_3^2\right) \cdot n + \left(x_1 + x_2^2\right)$.

We now exploit that for $a_1, a_2 \in \mathbb{N}$ and $b_1, b_2 \in \mathcal{S}_{>0}$ the term $n^{a_1} \cdot b_1^n$ asymptotically dominates $n^{a_2} \cdot b_2^n$ if $(b_1, a_1) >_{\text{lex}} (b_2, a_2)$, i.e., in a poly-exponential expression $\sum_{i=1}^{n} \alpha_i \cdot n^{a_i} \cdot b_i^n$ we can order the coefficients α_i according to the asymptotic growth of $n^{a_i} \cdot b_i^n$.

▶ **Example 3.** Continuing Ex. 2, the coefficients of pe are $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ where:

$$\begin{array}{lllllll} \alpha_1 & = & \frac{4}{3} \cdot x_3^5 & & \alpha_2 & = & -2 \cdot x_3^5 - 2 \cdot x_2 \cdot x_3^3 + 4 \cdot x_3^4 \\ \alpha_3 & = & x_2^2 \cdot x_3 + \frac{2}{3} \cdot x_3^5 + 2 \cdot x_2 \cdot x_3^3 - 4 \cdot x_2 \cdot x_3^2 & & \alpha_4 & = & x_1 + x_2^2 \end{array}$$

Here, α_1 is the coefficient of the asymptotically largest term, α_2 is the coefficient of the asymptotically second largest term, etc.

This allows us to reduce eventual non-termination to $\text{Th}_\exists(\mathcal{S}, \mathbb{R}_\mathbb{A})$ if φ is an atom: $\exists \vec{x} \in \mathcal{S}^d$, $n_0 \in \mathbb{N}$. $\forall n \in \mathbb{N}_{>n_0}$. $pe > 0$ is valid if there is an $\vec{e} \in \mathcal{S}^d$ such that the coefficient of the asymptotically dominant term in $pe(\vec{e})$ is positive. However, this coefficient is simply the coefficient α of the asymptotically largest term in pe with $\alpha(\vec{e}) \neq 0$ (similar for $pe \geq 0$).

▶ **Lemma 4.** *Given a poly-exponential expression pe and* $\triangleright \in \{\geq, >\}$*, one can reduce validity of* $\exists \vec{x} \in \mathcal{S}^d$, $n_0 \in \mathbb{N}$. $\forall n \in \mathbb{N}_{>n_0}$. $pe \triangleright 0$ *to validity of a closed formula from* $\mathrm{Th}_\exists(\mathcal{S}, \mathbb{R}_\mathbb{A})$.

▶ **Example 5.** We finish Ex. 3 for $\mathcal{S} = \mathbb{Z}$. Here, (3) is valid iff

$$\exists x_1, x_2, x_3 \in \mathbb{Z}. \bigvee_{i=1}^{4} \alpha_i > 0 \wedge \bigwedge_{j=1}^{i-1} (\alpha_j = 0)$$

is valid. Thus, $[x_1/-4, x_2/2, x_3/1]$ satisfies $\alpha_1 > 0$ as $\left(\frac{4}{3} \cdot 1^5\right) > 0$. So $(-4, 2, 1)$ witnesses eventual non-termination of (2), i.e., (2) does *not* terminate.

This result can be generalized to arbitrary propositional formulas (see [2]). By combining these insights with chaining, we finally get the following result.

▶ **Theorem 6** ((Semi-)Decidability of (Non-)Termination)**.** *Termination of twn-loops on* $\mathcal{S}^d$ *is reducible to* $\mathrm{Th}_\exists(\mathcal{S}, \mathbb{R}_\mathbb{A})$*. So for twn-loops, termination is decidable over* $\mathcal{S} = \mathbb{R}_\mathbb{A}$ *and non-termination is semi-decidable over* $\mathcal{S} \in \{\mathbb{Z}, \mathbb{Q}\}$.

4 The Halting Problem

We now consider the *halting problem* for *twn*-loops. In contrast to termination, i.e., to the question whether a loop terminates for *all* $\vec{e} \in \mathcal{S}^d$, the halting problem asks whether a loop terminates for a *given* $\vec{e} \in \mathcal{S}^d$. In this section, we sketch the proof of the following theorem. The full proof is given in [3].

▶ **Theorem 7** (Decidability of the Halting Problem)**.** *The halting problem for twn-loops is decidable.*

Clearly, if the halting problem is decidable for $\mathcal{S} = \mathbb{R}_\mathbb{A}$, then it is also decidable for all subrings of $\mathbb{R}_\mathbb{A}$. Thus, throughout this section, w.l.o.g. we restrict ourselves to loops over $\mathbb{R}_\mathbb{A}$.

We now show that for any $\vec{e} \in \mathbb{R}_\mathbb{A}^d$, it is decidable whether $\vec{e}$ is a witness for non-termination. As chaining preserves (non-)termination, w.l.o.g. we consider *tnn*-loops. The key idea is *stabilization*. A loop $(\varphi, \vec{u})$ stabilizes on $\vec{e}$ after n_0 iterations iff $\forall n \geq n_0$. $\varphi(\vec{q}[\vec{x}/\vec{e}]) \iff \varphi(\vec{q}[\vec{x}/\vec{e}, n/n_0])$, where again $\vec{q}$ is the closed form of $\vec{u}^n$. The smallest value n_0 for which $(\varphi, \vec{u})$ stabilizes on $\vec{e}$ is called the *stabilization threshold* of $(\varphi, \vec{u})$ on $\vec{e}$ (denoted $sth_{(\varphi,\vec{u})}(\vec{e})$).

For any poly-exponential expression pe and $\vec{e} \in \mathbb{R}_\mathbb{A}^d$, let $sth_{pe}(\vec{e})$ be the smallest value n_0 such that $\forall n \geq n_0$. $\mathrm{sign}\,(pe[\vec{x}/\vec{e}]) = \mathrm{sign}\,(pe[\vec{x}/\vec{e}, n/n_0])$, where for any $c \in \mathbb{R}$, we define $\mathrm{sign}\,(c) = 1$ if $c > 0$, $\mathrm{sign}\,(c) = -1$ if $c < 0$, and $\mathrm{sign}\,(0) = 0$. So if $\varphi(\vec{q}) \equiv pe \triangleright 0$ with $\triangleright \in \{\geq, >\}$, then we have $sth_{(\varphi,\vec{u})} \leq sth_{pe}$. Since every poly-exponential expression is weakly monotonic w.r.t. n for large enough values of n, every *tnn*-loop stabilizes on each $\vec{e} \in \mathbb{R}_\mathbb{A}^d$. Thus, it suffices to find a computable upper bound n_0 on $sth_{(\varphi,\vec{u})}(\vec{e})$ to decide the halting problem. Then $(\varphi, \vec{u})$ diverges on $\vec{e}$ iff $\forall n \leq n_0$. $\varphi(\vec{u}^n(\vec{e}))$.

To infer a bound on $sth_{pe}(\vec{e})$, note that $pe(\vec{e})$ only contains the variable n. Therefore, we can easily compute an upper bound on the stabilization threshold. This is due to the fact that for any $a_1, a_2 \in \mathbb{N}$, $b_1, b_2 \in (\mathbb{R}_\mathbb{A})_{>0}$ with $(b_1, a_1) >_{\text{lex}} (b_2, a_2)$, and $k \in \mathbb{R}_\mathbb{A}$, we can compute an $n_0 \in \mathbb{N}$ such that $n^{a_1} \cdot b_1^n > k \cdot n^{a_2} \cdot b_2^n$ for all $n \geq n_0$ (see [3] for details).

▶ **Lemma 8.** *For any poly-exponential expression pe and any* $\vec{e} \in \mathbb{R}_\mathbb{A}^d$*, one can compute an* $m \in \mathbb{N}$ *with* $m \geq sth_{pe}(\vec{e})$.

Iterating this computation for all inequations occurring in $\varphi(\vec{q})$ finishes the proof of Thm. 7.

▶ **Example 9.** Reconsider the setting from Ex. 2, 3, and 5. We show how to decide the halting problem for the witness $(-4, 2, 1)$ for eventual non-termination from Ex. 5: We have

$$pe[x_1/-4, x_2/2, x_3/1] = \tfrac{4}{3} \cdot n^3 - 2 \cdot n^2 + \tfrac{2}{3} \cdot n.$$

Here, $n_0 = 4$ is an upper bound on $sth_{pe}(-4, 2, 1)$ as for all $n \geq 4$ we have $n^3 > 2 \cdot n^2$. While $\bigwedge_{n=2}^{4} \left(\frac{4}{3} \cdot n^3 - 2 \cdot n^2 + \frac{2}{3} \cdot n > 0\right)$ holds, $\left(\frac{4}{3} \cdot 1^3 - 2 \cdot 1^2 + \frac{2}{3} \cdot 1 > 0\right)$ is false. Thus, $(-4, 2, 1)$ is not a witness for non-termination of (2). Still, we have just proven that $\vec{u}^2(-4, 2, 1) = (0, -2, 1)$ witnesses non-termination of (2).

5 Runtime Bounds

In the following, we restrict ourselves to *twn*-loops over the *integers* and show how to obtain upper bounds on their *runtime*. For a loop $(\varphi, \vec{u})$ over $\mathbb{Z}$, its *runtime* $rt_{(\varphi,\vec{u})}(\vec{e})$ on a *terminating* input $\vec{e} \in \mathbb{Z}^d$ is the smallest $n \in \mathbb{N}$ such that $\varphi(\vec{u}^n(\vec{e}))$ is false, i.e., $(\varphi, \vec{u})$ terminates on $\vec{e}$ after n iterations. In practice, it is usually infeasible to compute the runtime exactly, so that state-of-the-art complexity analyzers rely on approximations. In this spirit, we prove that the runtime of a *twn*-loop on $\mathbb{Z}^d$ is bounded (from above) by a polynomial in the 1-norm $\|\vec{x}\| = \sum_{j=1}^{d} |x_j|$ of $\vec{x}$. Moreover, this polynomial is *computable*.

▶ **Theorem 10** (Polynomial Loops Have Polynomial Runtime)**.** *Let $(\varphi, \vec{u})$ be a twn-loop over $\mathbb{Z}$ and let $\vec{q}$ be the closed form of $\vec{u}^n$. One can compute a polynomial $f \in \mathbb{N}[y]$ such that $rt_{(\varphi,\vec{u})}(\vec{e}) \leq f(\|\vec{e}\|)$ for all $\vec{e}$ on which $(\varphi, \vec{u})$ terminates. If $(\varphi, \vec{u})$ is linear, then f is linear.*

We restrict ourselves to loops over $\mathbb{Z}$ as measuring the "size" of non-integer inputs via the 1-norm is not suitable for analyzing runtime complexity (see [3]).

We give the idea of the proof of Thm. 10 and refer to [3] for the details. As we clearly have $rt_{(\varphi,\vec{u})}(\vec{e}) \leq sth_{(\varphi,\vec{u})}(\vec{e})$ for all $\vec{e}$ on which $(\varphi, \vec{u})$ terminates, to derive an upper bound on $rt_{(\varphi,\vec{u})}$, it suffices to find an upper bound on $sth_{(\varphi,\vec{u})}$.

Recall that the coefficients of the poly-exponential expressions in $\varphi(\vec{q})$ are polynomials from $\mathbb{Q}[\vec{x}]$, i.e., we can rescale these expressions such that all occurring coefficients are polynomials over the integers. Exploiting this fact, similar to Lemma 8 one can compute an upper bound on $sth_{(\varphi,\vec{u})}(\vec{e})$ depending on $\|\vec{e}\|$. Thus, this then provides a bound on the runtime of the loop on terminating inputs. This result is different from Lemma 8, where we computed an upper bound on $sth_{(\varphi,\vec{u})}(\vec{e})$ for a *fixed* input, whereas in this section we consider an upper bound on $sth_{(\varphi,\vec{u})}(\vec{e})$ for *arbitrary* values of $\vec{e}$.

▶ **Example 11.** We again regard the setting from Ex. 2, 3, and 5. Consider $3 \cdot pe$, i.e, we rescale by the only denominator 3 and thus, the coefficients of $3 \cdot pe$ are $3 \cdot \alpha_1, \ldots, 3 \cdot \alpha_4$.

Taking into account that for each $1 \leq i \leq 4$ we have $|3 \cdot \alpha_i(\vec{e})| \leq 12 \cdot \sum_{i=0}^{5} (|e_1| + |e_2| + |e_3|)^i$ we obtain $f(y) = 2 \cdot 12 \cdot \sum_{i=0}^{5} y^i + 3 = 24 \cdot \sum_{i=1}^{5} y^i + 27$. Here, the factor of 2 and the constant addend 3 stem from the formal proof of this result in [3] where we split sums and over-approximate each part separately. Thus, $sth_{pe}(e_1, e_2, e_3)$ is bounded by $f(\|(e_1, e_2, e_3)\|) = 24 \cdot \sum_{i=1}^{5} \|(e_1, e_2, e_3)\|^i + 27$ for all $(e_1, e_2, e_3) \in \mathbb{Z}^3$.

6 Related Work and Conclusion

We presented a reduction from termination of *twn*-loops to $\mathrm{Th}_{\exists}(\mathcal{S}, \mathbb{R}_{\mathbb{A}})$. This implies decidability of termination over $\mathcal{S} = \mathbb{R}_{\mathbb{A}}$ and semi-decidability of non-termination over $\mathcal{S} = \mathbb{Z}$ and $\mathcal{S} = \mathbb{Q}$. Moreover, we proved decidability of the halting problem over $\mathcal{S} = \mathbb{R}_{\mathbb{A}}$. Finally, we showed that bounds on the runtime of *twn*-loops over the integers can always be computed.

To see why complete approaches for termination and complexity are useful, consider

$$\textbf{while } x_1 \geq x_2 \wedge x_2 \geq 1 \textbf{ do } \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \leftarrow \begin{bmatrix} 2 \cdot x_1 \\ 3 \cdot x_2 \end{bmatrix}.$$

This *twn*-loop terminates, but proving it is beyond the capabilities of *lexicographic combinations of linear ranking functions* [4]. Such ranking functions represent the most popular approach to prove termination and to infer runtime bounds automatically. In contrast, our approach does not use ranking functions, but it reasons about the expression obtained from the loop condition when substituting the variables by the closed forms of their iterated updates. Thus, with our approach, one can show termination of the loop and compute the bound $2 \cdot (|x_1| + |x_2|) + 3$ on its runtime.

References

1 Amir M. Ben-Amram, Samir Genaim, and Abu Naser Masud. On the termination of integer loops. *ACM Transactions on Programming Languages and Systems*, 34(4), 2012. doi:10.1145/2400676.2400679.

2 Florian Frohn, Marcel Hark, and Jürgen Giesl. Termination of polynomial loops. In *Proc. SAS '20*, LNCS 12389, pages 89–112, 2020. Long version available at https://arxiv.org/abs/1910.11588. doi:10.1007/978-3-030-65474-0_5.

3 Marcel Hark, Florian Frohn, and Jürgen Giesl. Polynomial loops: Beyond termination. In *Proc. LPAR '20*, volume 73 of *EPiC Series in Computing*, pages 279–297, 2020. doi:10.29007/nxv1.

4 Jan Leike and Matthias Heizmann. Ranking templates for linear loops. *Logical Methods in Computer Science*, 11(1), 2015. doi:10.2168/LMCS-11(1:16)2015.

5 Joël Ouaknine, João Sousa Pinto, and James Worrell. On termination of integer linear loops. In *Proc. SODA 2015*, pages 957–969, 2015. doi:10.1137/1.9781611973730.65.

Polynomial Termination over $\mathbb{N}$ is Undecidable

Fabian Mitterwallner
University of Innsbruck, Innsbruck, Austria

Aart Middeldorp
University of Innsbruck, Innsbruck, Austria

Abstract

In this paper we prove that the problem whether the termination of a given rewrite system can be shown by a polynomial interpretation in the natural numbers is undecidable.

2012 ACM Subject Classification Theory of computation → Equational logic and rewriting; Theory of computation → Rewrite systems; Theory of computation → Computability

Keywords and phrases term rewriting, polynomial termination, undecidability

Acknowledgements We thank the reviewers for critically reading the paper, and providing comments and suggestions.

1 Introduction

Proving termination of a rewrite system by using a polynomial interpretation over the natural numbers goes back to Lankford [4]. Two problems need to be addressed when using polynomial interpretations for proving termination, whether by hand or by a tool:

1. finding suitable polynomials for the function symbols,
2. showing that the induced order constraints on polynomials are valid.

The latter problem amounts to ($\star$) proving $P(x_1, \ldots, x_n) > 0$ for all natural numbers $x_1, \ldots, x_n \in \mathbb{N}$, for polynomials $P \in \mathbb{Z}[x_1, \ldots, x_n]$. This is known to be undecidable, as a consequence of Hilbert's 10th Problem, see e.g., Zantema [6, Proposition 6.2.11]. Heuristics for the former problem are presented in [2, 6]. In this paper we prove the undecidability of the existence of a termination proof by a polynomial interpretation in $\mathbb{N}$ by a reduction from ($\star$). This result is not surprising, but we are not aware of a proof of undecidability in the literature, and the construction is not entirely obvious. We construct a family of rewrite systems $\mathcal{R}_P$ parameterized by polynomials $P \in \mathbb{Z}[x_1, \ldots, x_n]$ such that $\mathcal{R}_P$ is polynomially terminating over $\mathbb{N}$ if and only if $P(x_1, \ldots, x_n) > 0$ for all $x_1, \ldots, x_n \in \mathbb{N}$. The construction is based on techniques from [5], in which specific rewrite rules enforce the interpretations of certain function symbols.

2 Undecidability of Polynomial Termination

We assume familiarity with term rewriting [1], but recall the definition of polynomial termination over $\mathbb{N}$. Given a signature $\mathcal{F}$, a well-founded monotone $\mathcal{F}$-algebra $(\mathcal{A}, >)$ consists of a non-empty $\mathcal{F}$-algebra $\mathcal{A} = (A, \{f_\mathcal{A}\}_{f \in \mathcal{F}})$ and a well-founded order $>$ on the carrier A of $\mathcal{A}$ such that every algebra operation is strictly monotone in all its coordinates, i.e., if $f \in \mathcal{F}$ has arity $n \geqslant 1$ then $f_\mathcal{A}(a_1, \ldots, a_i, \ldots, a_n) > f_\mathcal{A}(a_1, \ldots, b, \ldots, a_n)$ for all $a_1, \ldots, a_n, b \in A$ and $i \in \{1, \ldots, n\}$ with $a_i > b$. The induced order $>_\mathcal{A}$ on terms is a reduction order that ensures the termination of any compatible (i.e., $\ell >_\mathcal{A} r$ for all rewrite rules $\ell \to r$) TRS $\mathcal{R}$. We call $\mathcal{R}$ *polynomially terminating over* $\mathbb{N}$ if compatibility holds when the underlying algebra $\mathcal{A}$ is restricted to the set of natural numbers $\mathbb{N}$ with standard order $>_\mathbb{N}$ such that every n-ary function symbol f is interpreted as a monotone polynomial $f_\mathbb{N}$ in $\mathbb{Z}[x_1, \ldots, x_n]$.

Table 1 The TRS $\mathcal{R}$.

$\mathsf{g}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{g}(x)))$	(A)	$\mathsf{s}(\mathsf{s}(0)) \to \mathsf{q}(\mathsf{s}(0))$	(G)
$\mathsf{q}(\mathsf{g}(x)) \to \mathsf{g}(\mathsf{g}(\mathsf{q}(x)))$	(B)	$\mathsf{s}(0) \to \mathsf{q}(0)$	(H)
$\mathsf{g}(x) \to \mathsf{a}(x,x)$	(C)	$\mathsf{q}(\mathsf{s}(0)) \to 0$	(I)
$\mathsf{s}(x) \to \mathsf{a}(0,x)$	(D)	$\mathsf{s}^5(0) \to \mathsf{q}(\mathsf{s}(\mathsf{s}(0)))$	(J)
$\mathsf{s}(x) \to \mathsf{a}(x,0)$	(E)	$\mathsf{q}(\mathsf{s}(\mathsf{s}(0))) \to \mathsf{s}^3(0)$	(K)
$\mathsf{a}(\mathsf{q}(x),\mathsf{g}(x)) \to \mathsf{q}(\mathsf{s}(x))$	(F)	$\mathsf{s}(\mathsf{a}(x,x)) \to \mathsf{d}(x)$	(L)
		$\mathsf{s}(\mathsf{d}(x)) \to \mathsf{a}(x,x)$	(M)
		$\mathsf{s}(\mathsf{a}(\mathsf{q}(\mathsf{a}(x,y)),\mathsf{d}(\mathsf{a}(x,y)))) \to \mathsf{a}(\mathsf{a}(\mathsf{q}(x),\mathsf{q}(y)),\mathsf{d}(\mathsf{m}(x,y)))$	(N)
		$\mathsf{s}(\mathsf{a}(\mathsf{a}(\mathsf{q}(x),\mathsf{q}(y)),\mathsf{d}(\mathsf{m}(x,y)))) \to \mathsf{a}(\mathsf{q}(\mathsf{a}(x,y)),\mathsf{d}(\mathsf{a}(x,y)))$	(O)

Whereas well-founded monotone algebras are complete for termination, polynomial termination gives rise to a much more restricted class of TRSs. For instance, Hofbauer and Lautemann [3] proved that polynomially terminating TRSs induce a double-exponential upper bound on the derivational complexity.

Our rewrite systems $\mathcal{R}_P$ consists of three parts: a fixed component $\mathcal{R}$, which is extended to $\mathcal{R}_n$ for some $n \in \mathbb{N}$ depending on the exponents in P, and a single rewrite rule that encodes the positiveness of P. For the latter we need function symbols that are interpreted as addition and multiplication. That is the purpose of the TRS $\mathcal{R}$, whose rules are presented in Table 1. It is a simplified and modified version of the TRS $\mathcal{R}_2$ in [5]. Since multiplication is not strictly monotone on $\mathbb{N}$, the rules (N) and (O) restrict the interpretation of m to $xy + x + y$, which suffices for the reduction.

▶ **Lemma 1.** *The* TRS $\mathcal{R}$ *is polynomially terminating over* $\mathbb{N}$.

Proof. The well-founded algebra $(\mathbb{N}, >_{\mathbb{N}})$ with interpretations

$$\mathsf{0}_{\mathbb{N}} = 0 \qquad \mathsf{s}_{\mathbb{N}}(x) = x + 1 \qquad \mathsf{a}_{\mathbb{N}}(x,y) = x + y \qquad \mathsf{q}_{\mathbb{N}}(x) = x^2$$
$$\mathsf{d}_{\mathbb{N}}(x) = 2x \qquad \mathsf{g}_{\mathbb{N}}(x) = 4x + 6 \qquad \mathsf{m}_{\mathbb{N}}(x,y) = xy + x + y$$

is monotone and compatible with $\mathcal{R}$. Hence $\mathcal{R}$ is polynomially terminating. ◀

Note that this polynomial interpretation is found by the termination tool TTT2 with the strategy `poly -direct -nl2 -ib 4 -ob 6`.

More importantly, to ensure termination in $(\mathbb{N}, >_{\mathbb{N}})$, the rewrite rules of $\mathcal{R}$ mandate that the interpretation of some of the function symbols is unique. The proof of the following lemma closely follows the reasoning in [5, Lemmata 4.4 and 5.2].

▶ **Lemma 2.** *Any monotone polynomial interpretation* $(\mathbb{N}, >_{\mathbb{N}})$ *compatible with* $\mathcal{R}$ *must interpret the function symbols* $\mathsf{0}$, s, d, a, m *and* q *as follows:*

$$\mathsf{0}_{\mathbb{N}} = 0 \qquad \mathsf{s}_{\mathbb{N}}(x) = x + 1 \qquad \mathsf{a}_{\mathbb{N}}(x,y) = x + y$$
$$\mathsf{d}_{\mathbb{N}}(x) = 2x \qquad \mathsf{m}_{\mathbb{N}}(x,y) = xy + x + y \qquad \mathsf{q}_{\mathbb{N}}(x) = x^2$$

Proof. Compatibility with (A) implies

$$\deg(\mathsf{g}_{\mathbb{N}}) \cdot \deg(\mathsf{s}_{\mathbb{N}}) \geqslant \deg(\mathsf{s}_{\mathbb{N}})^2 \cdot \deg(\mathsf{g}_{\mathbb{N}})$$

This is only possible if $\deg(\mathsf{s}_\mathbb{N}) \leqslant 1$. Together with the strict monotonicity of $\mathsf{s}_\mathbb{N}$ we obtain $\deg(\mathsf{s}_\mathbb{N}) = 1$. Hence s must be interpreted by a linear polynomial: $\mathsf{s}_\mathbb{N}(x) = s_1 x + s_0$ with $s_1 \geqslant 1$ and $s_0 \geqslant 0$. The same reasoning applied to (B) yields $\mathsf{g}_\mathbb{N}(x) = g_1 x + g_0$ for some $g_1 \geqslant 1$ and $g_0 \geqslant 0$. The compatibility constraint imposed by rule (A) further gives rise to the inequality

$$g_1 s_1 x + g_1 s_0 + g_0 > g_1 s_1^2 x + g_0 s_1^2 + s_1 s_0 + s_0 \tag{1}$$

for all $x \in \mathbb{N}$. Since $s_1 \geqslant 1$ and $g_1 \geqslant 1$, this only holds if $s_1 = 1$. Simplifying (1) we obtain $g_1 s_0 > 2 s_0$, which implies $s_0 > 0$ and $g_1 > 2$. If $\mathsf{q}_\mathbb{N}$ were linear, the same reasoning could be applied to (B) resulting in $g_1 = 1$, contradicting $g_1 > 2$. Hence $\mathsf{q}_\mathbb{N}$ is at least quadratic.

Next we turn our attention to the rewrite rules (C) – (F). Because $\mathsf{g}_\mathbb{N}$ is linear, compatibility with (C) and strict monotonicity of $\mathsf{a}_\mathbb{N}$ ensures $\deg(\mathsf{a}_\mathbb{N}) = 1$. Hence, $\mathsf{a}_\mathbb{N} = a_2 x + a_1 y + a_0$ with $a_2 \geqslant 1$, $a_1 \geqslant 1$ and $a_0 \geqslant 0$. From compatibility with rules (D) and (E) we obtain $a_1 = 1$ and $a_2 = 1$. Using the current shapes of $\mathsf{a}_\mathbb{N}$, $\mathsf{g}_\mathbb{N}$ and $\mathsf{s}_\mathbb{N}$, compatibility with rule (F) yields the inequality $\mathsf{g}_\mathbb{N}(x) + a_0 > \mathsf{q}_\mathbb{N}(x + s_0) - \mathsf{q}_\mathbb{N}(x)$ for all $x \in \mathbb{N}$. This can only be the case if $\deg(\mathsf{g}_\mathbb{N}(x) + a_0) \geqslant \deg(\mathsf{q}_\mathbb{N}(x + s_0) - \mathsf{q}_\mathbb{N}(x))$, which in turn simplifies to $1 \geqslant \deg(\mathsf{q}_\mathbb{N}(x)) - 1$. Hence $\mathsf{q}_\mathbb{N}(x) = q_2 x^2 + q_1 x + q_0$ with $q_2 \geqslant 1$. From monotonicity we also have $\mathsf{q}_\mathbb{N}(1) > \mathsf{q}_\mathbb{N}(0)$, which leads to $q_2 + q_1 \geqslant 1$.

To further constrain $\mathsf{s}_\mathbb{N}$ we consider the rewrite rule (G). The compatibility constraint gives rise to

$$\begin{aligned}
\mathsf{0}_\mathbb{N} + 2 s_0 &> q_2 (\mathsf{0}_\mathbb{N} + s_0)^2 + q_1 (\mathsf{0}_\mathbb{N} + s_0) + q_0 \\
&= q_2 \mathsf{0}_\mathbb{N}^2 + q_2 s_0^2 + \mathsf{0}_\mathbb{N}(2 q_2 s_0 + q_1) + q_1 s_0 + q_0 \\
&\geqslant q_2 s_0^2 + \mathsf{0}_\mathbb{N} + (1 - q_2) s_0 && (q_2 + q_1 \geqslant 1 \text{ and } q_0, q_2, s_0 \geqslant 1) \\
&= q_2 s_0 (s_0 - 1) + \mathsf{0}_\mathbb{N} + s_0 \\
&\geqslant s_0^2 + \mathsf{0}_\mathbb{N} && (s_0 \geqslant 1)
\end{aligned}$$

Hence the inequality $2 s_0 > s_0^2$ holds, which is only true if $s_0 = 1$. Therefore $\mathsf{s}_\mathbb{N}(x) = x + 1$. Compatibility with (D) now amounts to $x + 1 > \mathsf{0}_\mathbb{N} + x + a_0$, which implies $\mathsf{0}_\mathbb{N} = a_0 = 0$. At this point we have uniquely constrained $\mathsf{0}_\mathbb{N}$, $\mathsf{s}_\mathbb{N}$ and $\mathsf{a}_\mathbb{N}$. To fully constrain $\mathsf{q}_\mathbb{N}$ we turn to (H), which implies $q_0 = 0$, the rules (G) and (I), which together imply $2 > \mathsf{q}_\mathbb{N}(1) > 0$ and thus $\mathsf{q}_\mathbb{N}(1) = q_2 + q_1 = 1$, and the rules (J) and (K), which imply $5 > \mathsf{q}_\mathbb{N}(2) > 3$ and thus $\mathsf{q}_\mathbb{N}(2) = 4 q_2 + 2 q_1 = 4$. Consequently, $q_2 = 1$ and $q_1 = 0$. Hence $\mathsf{q}_\mathbb{N}(x) = x^2$. Compatibility with the rules (L) and (M) yields $x + x + 1 > \mathsf{d}_\mathbb{N}(x)$ and $\mathsf{d}_\mathbb{N}(x) + 1 > x + x$ which imply $\mathsf{d}_\mathbb{N}(x) = 2x$. Finally, compatibility with the rules (N) and (O) amounts to $(x + y)^2 + 2x + 2y + 1 > x^2 + y^2 + 2\,\mathsf{m}_\mathbb{N}(x, y) \geqslant (x + 1)^2 + 2x + 2y$, which uniquely determines $\mathsf{m}_\mathbb{N}(x, y) = xy + x + y$. ◀

Using the previously fixed interpretations we can now add new function symbols, and more easily mandate their interpretations. By adding the two rules

$$\mathsf{s}(t) \to u \qquad\qquad \mathsf{s}(u) \to t$$

for some terms t and u, we enforce an equality constraint on the interpretations of t and u, assuming the system remains polynomially terminating.

To represent the exponents in the polynomial P we add symbols p_i for $1 \leqslant i \leqslant n$, where n is the maximal exponent in P. To fix $(\mathsf{p}_i)_\mathbb{N}(x) = x^i$, we add two rules per symbol, according to the following definition.

▶ **Definition 3.** *We define a family of* TRS*s* $(\mathcal{R}_n)_{n\geqslant 0}$ *as follows:*

$$\begin{aligned}\mathcal{R}_0 &= \mathcal{R}\\ \mathcal{R}_1 &= \mathcal{R}_0 \cup \{\mathsf{s}(\mathsf{p}_1(x)) \to x,\ \mathsf{s}(x) \to \mathsf{p}_1(x)\}\\ \mathcal{R}_{n+1} &= \mathcal{R}_n \cup \left\{\begin{aligned}\mathsf{s}(\mathsf{a}(\mathsf{p}_{n+1}(x), \mathsf{a}(x, \mathsf{p}_n(x)))) &\to \mathsf{m}(x, \mathsf{p}_n(x))\\ \mathsf{s}(\mathsf{m}(x, \mathsf{p}_n(x))) &\to \mathsf{a}(\mathsf{p}_{n+1}(x), \mathsf{a}(x, \mathsf{p}_n(x)))\end{aligned}\right\}\end{aligned}$$

▶ **Lemma 4.** *For any* $n \geqslant 0$*, the* TRS $\mathcal{R}_n$ *is polynomially terminating over* $\mathbb{N}$ *if and only if* $(\mathsf{p}_i)_{\mathbb{N}}(x) = x^i$ *for all* $1 \leqslant i \leqslant n$*.*

Proof. From Lemma 1 we know that $\mathcal{R}$ is polynomially terminating and the interpretations are unique due to Lemma 2. Hence the Lemma holds for $\mathcal{R}_0$. For $n \geqslant 1$, the *if* direction holds, since the interpretations $(\mathsf{p}_i)_{\mathbb{N}}$ are monotone and the polynomial interpretation is compatible with $\mathcal{R}_n$:

$$x + 1 > x \qquad\qquad x + 1 > x$$

for $\mathcal{R}_1 \setminus \mathcal{R}_0$ and

$$x^n + x + x^{n-1} + 1 > xx^{n-1} + x + x^{n-1} \qquad xx^{n-1} + x + x^{n-1} + 1 > xx^n + x + x^{n-1}$$

for $\mathcal{R}_n \setminus \mathcal{R}_{n-1}$. For the *only if* direction we show that compatibility with the additional rules implies $(\mathsf{p}_i)_{\mathbb{N}}(x) = x^i$ for all $1 \leqslant i \leqslant n$. This is done by induction on n. For $n = 1$ the two rules in $\mathcal{R}_1 \setminus \mathcal{R}$ enforce $(\mathsf{p}_i)_{\mathbb{N}}(x) + 1 > x$ and $x + 1 > (\mathsf{p}_i)_{\mathbb{N}}(x)$. Hence $(\mathsf{p}_i)_{\mathbb{N}}(x) = x$. For $n > 1$ the rules in $\mathcal{R}_n \setminus \mathcal{R}_{n-1}$ enforce $(\mathsf{p}_n)_{\mathbb{N}}(x) = x \cdot (\mathsf{p}_{n-1})_{\mathbb{N}}(x)$ by the same reasoning. From the induction hypothesis we obtain $(\mathsf{p}_{n-1})_{\mathbb{N}}(x) = x^{n-1}$ and hence $(\mathsf{p}_n)_{\mathbb{N}} = x^n$ as desired. ◀

The fixed interpretations can now be used to construct arbitrary polynomials. Since non-monotone operations, such as subtraction (negative coefficients) and multiplication, cannot serve as interpretations for function symbols, we model these using the difference of two terms. In the following we write $[t]_{\mathbb{N}}$ for the polynomial that is the interpretation of the term t, according to the interpretations stated in Lemmata 2 and 4.

▶ **Lemma 5.** *For any monomial* $M = cx_1^{i_1} \cdots x_m^{i_m}$ *with* $i_1, \ldots, i_m > 0$ *and* $c \neq 0$ *there exist terms* ℓ_M *and* r_M *over the signature of* $\mathcal{R}_{\mathsf{max}(0,i_1,\ldots,i_m)}$*, such that* $M = [\ell_M]_{\mathbb{N}} - [r_M]_{\mathbb{N}}$ *and* $\mathcal{V}\mathsf{ar}(\ell_M) = \mathcal{V}\mathsf{ar}(r_M)$*.*

Proof. First we assume the coefficient c is positive. We construct ℓ_M and r_M by induction on m. If $m = 0$ then $M = c$ and we take $\ell_M = \mathsf{s}^c(\mathsf{0})$ and $r_M = \mathsf{0}$. We trivially have $\mathcal{V}\mathsf{ar}(\ell_M) = \varnothing = \mathcal{V}\mathsf{ar}(r_M)$ and $[\ell_M]_{\mathbb{N}} - [r_M]_{\mathbb{N}} = c - 0 = M$. For $m > 0$ we have $M = M' x_m^{i_m}$ with $M' = cx_1^{i_1} \cdots x_{m-1}^{i_{m-1}}$. The induction hypothesis yields terms $\ell_{M'}$ and $r_{M'}$ with $M' = [\ell_{M'}]_{\mathbb{N}} - [r_{M'}]_{\mathbb{N}}$ and $\mathcal{V}\mathsf{ar}(\ell_{M'}) = \mathcal{V}\mathsf{ar}(r_{M'})$. Hence

$$\begin{aligned}M &= M' x_m^{i_m} = [\ell_{M'}]_{\mathbb{N}} x_m^{i_m} - [r_{M'}]_{\mathbb{N}} x_m^{i_m}\\ &= (\mathsf{m}_{\mathbb{N}}([\ell_{M'}]_{\mathbb{N}}, x_m^{i_m}) - [\ell_{M'}]_{\mathbb{N}} - x_m^{i_m}) - (\mathsf{m}_{\mathbb{N}}([r_{M'}]_{\mathbb{N}}, x_m^{i_m}) - [r_{M'}]_{\mathbb{N}} - x_m^{i_m})\\ &= (\mathsf{m}_{\mathbb{N}}([\ell_{M'}]_{\mathbb{N}}, (\mathsf{p}_j)_{\mathbb{N}}(x_m)) + [r_{M'}]_{\mathbb{N}}) - (\mathsf{m}_{\mathbb{N}}([r_{M'}]_{\mathbb{N}}, (\mathsf{p}_j)_{\mathbb{N}}(x_m)) + [\ell_{M'}]_{\mathbb{N}})\end{aligned}$$

and thus we can take $\ell_M = \mathsf{a}(\mathsf{m}(\ell_{M'}, \mathsf{p}_j(x_m)), r_{M'})$ and $r_M = \mathsf{a}(\mathsf{m}(r_{M'}, \mathsf{p}_j(x_m)), \ell_{M'})$. Note that $\mathcal{V}\mathsf{ar}(\ell_M) = \mathcal{V}\mathsf{ar}(\ell_{M'}) \cup \{x_m\} \cup \mathcal{V}\mathsf{ar}(r_{M'}) = \mathcal{V}\mathsf{ar}(r_M)$.

If $c < 0$ then we take $\ell_M = r_{-M}$ and $r_M = \ell_{-M}$. We obviously have $\mathcal{V}\mathsf{ar}(\ell_M) = \mathcal{V}\mathsf{ar}(r_{-M}) = \mathcal{V}\mathsf{ar}(\ell_{-M}) = \mathcal{V}\mathsf{ar}(r_M)$. Moreover, $M = -(-M) = -([\ell_{-M}]_{\mathbb{N}} - [r_{-M}]_{\mathbb{N}}) = -([r_M]_{\mathbb{N}} - [\ell_M]_{\mathbb{N}}) = [\ell_M]_{\mathbb{N}} - [r_M]_{\mathbb{N}}$. ◀

▶ **Definition 6.** *Let $P = M_1 + \cdots + M_{k-1} + M_k \in \mathbb{Z}[x_1, \ldots, x_n]$ be a sum of monomials. We denote by ℓ_P the term $\mathsf{a}(\ell_1, \cdots \mathsf{a}(\ell_{k-1}, \ell_k) \cdots)$ and by r_P the term $\mathsf{a}(r_1, \cdots \mathsf{a}(r_{k-1}, r_k) \cdots)$. Here ℓ_i and r_i are the terms from applying Lemma 5 to M_i for $1 \leqslant i \leqslant k$. Moreover, $\ell_0 = r_0 = \mathsf{0}$. We define the* TRS $\mathcal{R}_P$ *as the extension of $\mathcal{R}_n$ with the single rule $\ell_P \to r_P$. Here n is the maximal exponent occurring in P.*

Note that the rewrite rule $\ell_P \to r_P$ in $\mathcal{R}_P$ is well-defined; ℓ_P is not a variable and $\mathcal{V}\mathsf{ar}(\ell_P) = \mathcal{V}\mathsf{ar}(r_P)$ as a consequence of Lemma 5.

▶ **Example 7.** The polynomial $P = 2x^2y - xy + 3$ is first split into its monomials $M_1 = 2x^2y$, $M_2 = -xy$ and $M_3 = 3$. Hence we obtain the TRS $\mathcal{R}_{P_1} = \mathcal{R}_2 \cup \{\mathsf{a}(\ell_{M_1}, \mathsf{a}(\ell_{M_2}, \ell_{M_3})) \to \mathsf{a}(r_{M_1}, \mathsf{a}(r_{M_2}, r_{M_3}))\}$, where

$$\ell_{M_1} = \mathsf{a}(\mathsf{m}(\underbrace{\mathsf{a}(\mathsf{m}(\mathsf{s}^2(\mathsf{0}), \mathsf{p}_2(x)), \mathsf{0})}_{\ell_{2x^2}}, \mathsf{p}_1(y)), \underbrace{\mathsf{a}(\mathsf{m}(\mathsf{0}, \mathsf{p}_2(x)), \mathsf{s}^2(\mathsf{0}))}_{r_{2x^2}})$$

$$r_{M_1} = \mathsf{a}(\mathsf{m}(\underbrace{\mathsf{a}(\mathsf{m}(\mathsf{0}, \mathsf{p}_2(x)), \mathsf{s}^2(\mathsf{0}))}_{r_{2x^2}}, \mathsf{p}_1(y)), \underbrace{\mathsf{a}(\mathsf{m}(\mathsf{s}^2(\mathsf{0}), \mathsf{p}_2(x)), \mathsf{0})}_{\ell_{2x^2}})$$

$$\ell_{M_2} = \mathsf{a}(\mathsf{m}(\underbrace{\mathsf{a}(\mathsf{m}(\mathsf{0}, \mathsf{p}_1(x)), \mathsf{s}(\mathsf{0}))}_{r_x}, \mathsf{p}_1(y)), \underbrace{\mathsf{a}(\mathsf{m}(\mathsf{s}(\mathsf{0}), \mathsf{p}_1(x)), \mathsf{0})}_{\ell_x})$$

$$r_{M_2} = \mathsf{a}(\mathsf{m}(\underbrace{\mathsf{a}(\mathsf{m}(\mathsf{s}(\mathsf{0}), \mathsf{p}_1(x)), \mathsf{0})}_{\ell_x}, \mathsf{p}_1(y)), \underbrace{\mathsf{a}(\mathsf{m}(\mathsf{0}, \mathsf{p}_1(x)), \mathsf{s}(\mathsf{0}))}_{r_x})$$

$$\ell_{M_3} = \mathsf{s}^3(\mathsf{0}) \qquad r_{M_3} = \mathsf{0}$$

Note that in the terms ℓ_{M_2} and r_{M_2} the ℓ and r of the recursive call are switched since M_2 has a negative coefficient.

▶ **Theorem 8.** *For any polynomial $P \in \mathbb{Z}[x_1, \ldots, x_n]$, the* TRS $\mathcal{R}_P$ *is polynomially terminating over $\mathbb{N}$ if and only if $P(x_1, \ldots, x_n) > 0$ for all $x_1, \ldots, x_n \in \mathbb{N}$.*

Proof. First suppose $\mathcal{R}_P$ is polynomially terminating over $\mathbb{N}$. So there exists a monotone polynomial interpretation in $(\mathbb{N}, >)$ that orients the rules of $\mathcal{R}_P$ from left to right. Let n be the maximum exponent in P. From Lemma 2 and Lemma 4 we infer that the interpretations of the function symbols $\mathsf{0}$, s, a, m, and p_i for $1 \leqslant i \leqslant n$ are fixed such that, according to Lemma 5, $P = [\ell_P]_{\mathbb{N}} - [r_P]_{\mathbb{N}}$. Since the rule $\ell_P \to r_P$ belongs to $\mathcal{R}_P$, $P(x_1, \ldots, x_n) > 0$ for all $x_1, \ldots, x_n \in \mathbb{N}$ by compatibility.

For the if direction, we assume that $P \in \mathbb{Z}[x_1, \ldots, x_n]$ satisfies $P(x_1, \ldots, x_n) > 0$ for all $x_1, \ldots, x_n \in \mathbb{N}$. By construction of $\ell_P \to r_P$ and Lemma 5, the interpretations in Lemma 2 and Lemma 4 orient the rule $\ell_P \to r_P$ from left to right. The same holds for rules $\mathcal{R}_n$. Hence $\mathcal{R}_P$ is polynomially terminating over $\mathbb{N}$. ◀

▶ **Corollary 9.** *It is undecidable whether a finite* TRS *is polynomially terminating over $\mathbb{N}$.*

3 Conclusion

We proved the undecidability of polynomial termination over the natural numbers, by a reduction from a variant of Hilbert's 10th problem. This was done by constructing a TRS R_P, for any polynomial $P \in \mathbb{Z}[x_1, \ldots, x_n]$, which can be shown to be polynomially terminating if and only if $P(x_1, \ldots, x_n) > 0$ for all $x_1, \ldots, x_n \in \mathbb{N}$. To construct this system we used techniques from [5] to fix the interpretation of function symbols. Using the fixed interpretations we constructed two terms ℓ_P and r_P, such that $P = [\ell_P]_{\mathbb{N}} - [r_P]_{\mathbb{N}}$. This

allowed us to encode the inequality $P > 0$ as the compatibility constraint associated with the rule $\ell_P \to r_P$.

In our proof we allow interpretations to be polynomials with integer coefficients. However, it equally applies if interpretations are limited to natural number coefficients, since the construction stays the same. We conclude the paper by mentioning two open questions.

1. Is polynomial termination over $\mathbb{N}$ decidable for terminating TRSs?

   The construction in this paper may produce non-terminating systems. Take for example the polynomial $P_1 = -1$. The resulting TRS $\mathcal{R}_{P_1} = \mathcal{R} \cup \{\mathsf{0} \to \mathsf{s}(\mathsf{0})\}$ is obviously not terminating.

2. Is incremental polynomial termination over $\mathbb{N}$, where we take the lexicographic extension of the order induced by the polynomial interpretations, decidable?

   We expect the answer is negative, but the construction in this paper needs to be modified. Consider for instance the polynomial $P_2 = x$. We obtain $\ell_{P_2} = \mathsf{a}(\mathsf{m}(\mathsf{s}(\mathsf{0}), \mathsf{p}_1(x)), \mathsf{0})$ and $r_{P_2} = \mathsf{a}(\mathsf{m}(\mathsf{0}, \mathsf{p}_1(x)), \mathsf{s}(\mathsf{0}))$. As a result, the TRS $\mathcal{R}_{P_2}$ is not polynomially terminating since $[\ell_{P_2}]_{\mathbb{N}} = 2x + 1 \not> x + 1 = [r_{P_2}]_{\mathbb{N}}$ for $x = 0$. However, if we take a second algebra $\mathcal{A}$ over $\mathbb{N}$ where the interpretation of m is changed to $\mathsf{m}_{\mathcal{A}}(x, y) = 2x + y$, then $[\ell_{P_2}]_{\mathcal{A}} = x + 2 > x + 1 = [r_{P_2}]_{\mathcal{A}}$ for all $x \in \mathbb{N}$. Hence the lexicographic order $(>_{\mathbb{N}}, >_{\mathcal{A}})$ is a reduction order compatible with $\mathcal{R}_{P_2}$.

References

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

2 Ahlem Ben Cherifa and Pierre Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, 9(2):137–159, 1987. doi:10.1016/0167-6423(87)90030-X.

3 Dieter Hofbauer and Clemens Lautemann. Termination proofs and the length of derivations (preliminary version). In *Proceedings of the 3rd International Conference on Rewriting Techniques and Applications*, volume 355 of *Lecture Notes in Computer Science*, pages 167–177, 1989. doi:10.1007/3-540-51081-8_107.

4 Dallas Lankford. On proving term rewrite systems are noetherian. Technical Report MTP-3, Louisiana Technical University, Ruston, LA, USA, 1979.

5 Friedrich Neurauter and Aart Middeldorp. Polynomial interpretations over the natural, rational and real numbers revisited. *Logical Methods in Computer Science*, 10(3:22):1–28, 2014. doi:10.2168/LMCS-10(3:22)2014.

6 Hans Zantema. Termination. In *Term Rewriting Systems*, chapter 6, pages 181–259. Cambridge University Press, 2003.

Modular Termination Analysis of C Programs (Extended Abstract)

Frank Emrich
Laboratory for Foundations of Computer Science, The University of Edinburgh, UK

Jera Hensel
LuFG Informatik 2, RWTH Aachen University, Aachen, Germany

Jürgen Giesl
LuFG Informatik 2, RWTH Aachen University, Aachen, Germany

Abstract

Termination analysis of C programs is challenging. On the one hand, the analysis needs to be precise. On the other hand, programs in practice are usually large and require substantial abstraction. In this extended abstract, we sketch an approach for modular symbolic execution to analyze termination of C programs with several functions. This approach is also suitable to handle recursive programs. We implemented it in our automated termination prover AProVE and evaluated its power on recursive and large programs.

2012 ACM Subject Classification Theory of computation → Semantics and reasoning

Keywords and phrases Modular Termination Analysis, Pointer Arithmetic, Recursion, C, LLVM

Funding funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 235950644 (Project GI 274/6-2)

1 Introduction

AProVE [7] is a tool for termination and complexity analysis of many programming languages including C. Its approach for termination analysis of C programs (which is illustrated in Fig. 1) focuses in particular on the connection between memory addresses and their contents. To avoid handling all intricacies of C, we use the Clang compiler [3] to transform programs into the platform-independent intermediate representation of the LLVM Compilation Framework [11]. As we presented in [12], in the first step our technique constructs a *symbolic execution graph* (SEG) which over-approximates all possible program runs and models memory addresses and contents explicitly. As a prerequisite for termination, AProVE shows the absence of undefined behavior during the construction of the SEG. In this way, our approach also proves memory safety of the program. Afterwards, the cycles of the graph are transformed into integer transition systems (ITSs) whose termination implies termination of the original C program. We use standard techniques to analyze termination of the ITSs, which are implemented in a back-end that AProVE also uses for termination analysis of other programming languages. Here, the satisfiability checkers Z3 [4], Yices [5], and MiniSAT [6] are applied to solve the search problems that arise during the termination proof.

Sometimes, the SEG does not contain over-approximating steps. Then, non-termination of the ITS resulting from a cycle of the graph together with a path from the root of the graph to the respective cycle implies non-termination of the program. In this case, our approach can also prove non-termination of C programs [9] by using the tool T2 [2] to show non-termination of ITSs. (AProVE's own back-end does not support the analysis of ITSs where runs may only begin with designated start terms.) While integers were considered to be unbounded in [12], we extended our approach to handle bitvector arithmetic and also

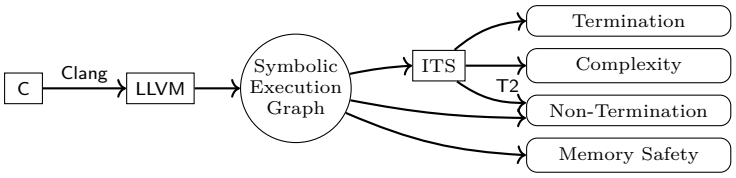


Figure 1 AProVE's workflow to prove termination and memory safety of C programs

discussed the use of our approach for complexity analysis of C programs in [10].

We showed how our approach supports programs with several functions in [12], but up to now it could not analyze functions in a modular way and it could not deal with recursion.

In this extended abstract, we sketch an idea on how to extend this approach to also support the abstraction of call stacks, which allows us to re-use previous analyses of auxiliary functions in a modular way. Moreover, in this way we can analyze recursive programs as well.

Another variation of our approach from [12] that also abstracts from call stacks was presented in [8]. Here, such an abstraction was necessary, because [8] defined abstract states such that they only contain a single stack frame. Thus, their variant can be seen as a first step towards the modularization of function graphs and the handling of recursion, but in contrast to us, the variant of [8] is restricted to integer programs without memory access. Moreover, the restriction to abstract states with a single stack frame prohibits the handling of non-tail recursive programs where the result of the recursive call is needed for the remaining computation.

Our technique for abstracting from the exact shape of the call stack in the symbolic execution graph is based on our earlier approach for termination analysis of Java Bytecode in [1], but the challenge is to adapt this idea to the byte-accurate representation of the memory needed for the analysis of C programs. A paragraph with a preliminary announcement of an extension of our approach to recursion was given in [9] and a full paper on our work has been submitted and is currently under review.

2 Modular Re-Use of Symbolic Execution Graphs

We presented an approach for symbolic execution of LLVM programs in [12] which computes a symbolic execution graph to capture all program runs. The states of the graph are so-called *abstract* program states that represent sets of concrete program states. We distinguish three types of edges in the graph:

(a) To execute the next instruction, we use *evaluation edges*. Based on formal rules that correspond to the actual execution of the respective LLVM instruction, a successor state is computed.

(b) Sometimes, the program flow depends on a condition whose truth value is not the same for all concrete states represented by the abstract state A to be evaluated. Then, we use a refinement rule to create two successors, where in the first one the condition holds and in the second, the negated condition holds. Now, A is connected by *refinement edges* to each of its successor states.

(c) Whenever we re-visit a program position that we had already reached earlier, we create a *generalization* state G, which represents all concrete program states that are represented by the previous abstract states at this program position in the graph. This step is crucial to obtain a finite symbolic execution graph. Then, we only keep the state A where the

program position was reached first in the graph and add a *generalization edge* from A to G. Now, evaluation continues from the generalized state G.

In [12], we defined the corresponding rules for the construction of the graph and proved their correctness. Moreover, we presented a generalization technique which guaranteed termination of the graph construction in case of non-recursive functions.

In the approach of [12], we distinguished between global knowledge that holds for the whole state (e.g., global variables and memory allocated by `malloc`), and local information stored in stack frames. Whenever a function `f` called an auxiliary function `g`, then during the construction of `f`'s symbolic execution graph, one obtained a new abstract state whose topmost stack frame is at the start of the function `g`. To evaluate this state further, now one had to execute `g` symbolically and only after the end of `g`'s execution, one could remove the topmost stack frame and continue the further execution of `f`. Even if one had analyzed termination of `g` before, in [12] one could not re-use its symbolic execution graph, but one had to perform a new symbolic execution of `g` whenever it is called. This missing modularity had severe drawbacks for the performance of the approach and moreover, it prevented the analysis of functions with recursive calls.

We now sketch an idea on how to abstract from the call stack by using call abstractions and intersections. This allows us to re-use previously computed symbolic execution graphs of auxiliary functions. Thus, it is the key for the modularization of our approach. To ease readability, in Fig. 2 we displayed abstract states only by a stack of program positions representing their call stack, and omitted all other components of the abstract states such as local and global variables and their values.

To prove termination of the function `f`, we start with a state A whose program position is at `f`'s initial instruction. If A evaluates (via several execution steps) to a state B where the function `g` is called, this yields a next state C where a new stack frame at `g`'s initial instruction is added on top of the stack of B (we refer to C as a *call state*). For a modular analysis of `g`, we perform *call abstraction*, which leads to a state D that results from C by removing all lower stack frames except the top one. In this way, we can compute a symbolic execution graph of `g` which is independent of the context that we have in `f`, so it is re-usable for other calls of `g`. Moreover, in case of recursive functions, this abstraction step is essential to ensure termination of the graph construction. Whenever the function `g` evaluates to a return state R where the function terminates, we have to take into account that the call of `g` in state C might lead to such a return state. Thus, for every pair of a call state C and a return state R of `g`, we construct an *intersection* state I which represents those states that result from C after completely executing the call of `g` in its topmost stack frame. This intersection state combines the stack frames of `f` and `g` in one state by integrating the results of `g`'s symbolic execution into the context of the call state C. This integration has to be done in such a way that on the one hand, all concrete states corresponding to a program execution along a path to I are really represented by I, and on the other hand, we keep as much knowledge as possible in I to find meaningful invariants for later termination proofs. Thus, we essentially have to compute the intersection of the knowledge in the states C and R. However, this is quite intricate when pointers are

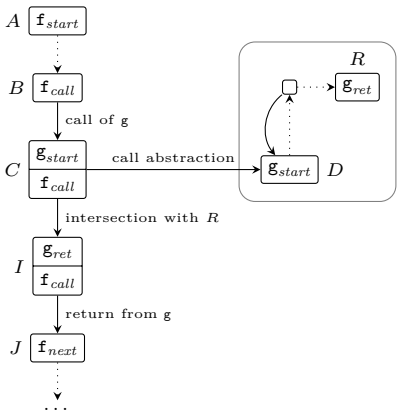


Figure 2 Modularized symbolic execution graph

involved that are reachable from different functions or that are passed by recursive function calls. In the example graph of Fig. 2, now the execution of `g` in the function `f` is represented by the path from B to J.

To summarize, we extend the symbolic execution graph construction by the following rules:

(d) Whenever a function is called within another function, the call state must have an outgoing *call abstraction edge* to its call abstraction state, which abstracts from all but the topmost stack frame and the knowledge related to it.
(e) For each call state C and each return state R of the same function, we create an abstract state I that represents the case that the execution of the topmost stack frame of C ended in R and should now continue with the lower stack frames of C. We call I the *intersection* of C and R, and each call state C has *intersection edges* to all its intersections. Intuitively, here the process of abstracting from lower stack frames is reversed in order to continue the execution with the former lower stack frames after returning to them.

Once we have a complete symbolic execution graph for the program under consideration, we extract *integer transition systems* (ITSs) from its maximal cycles and apply existing techniques to prove their termination. An ITS is a graph whose nodes are abstract states and whose edges are *transitions*. A transition is labeled with conditions that are required for its application. We use the same translation of symbolic execution graphs into ITSs that we already presented in [12].

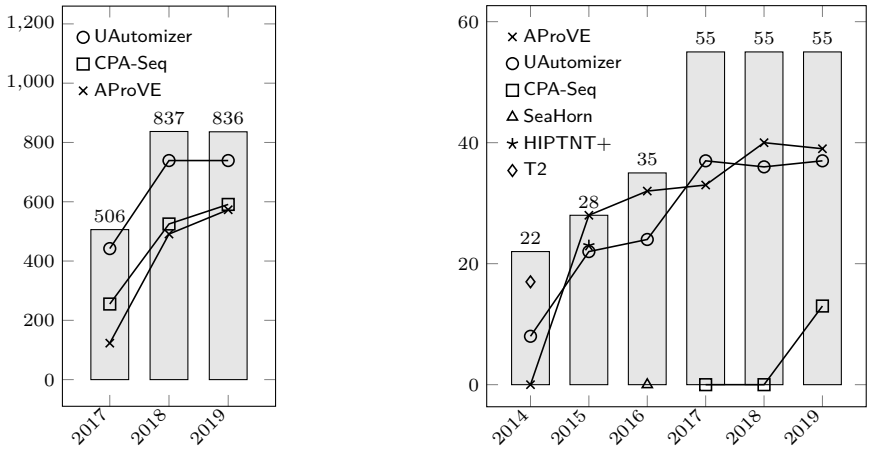
Our new modular approach does not only allow us to re-use the SEGs for auxiliary functions like `g` when they are called by other functions like `f`, but we also benefit from this modularity when extracting ITSs from the cycles of the symbolic execution graph. The reason is that the subgraphs of `f` and `g` do not share any cycles. Therefore, one can handle the ITSs extracted from these subgraphs completely independently when proving their termination, and for ITSs from re-used subgraphs, one does not have to prove termination again.

As in [12, Thm. 13] where we proved the correctness of our symbolic execution w.r.t. the formal definition of the LLVM semantics from the Vellvm project [13], our construction ensures that termination of the resulting ITSs implies termination of the original program.

▶ **Theorem 1** (Termination). *Let $\mathcal{P}$ be an* LLVM *program with a symbolic execution graph $\mathcal{G}$ and let $\mathcal{I}_1, \ldots, \mathcal{I}_m$ be the ITSs resulting from the cycles of $\mathcal{G}$. If all ITSs $\mathcal{I}_1, \ldots, \mathcal{I}_m$ terminate, then $\mathcal{P}$ also terminates for all concrete states c that are represented by a state of $\mathcal{G}$.*

3 Evaluation, Conclusion, and Future Work

We implemented our approach in the tool AProVE [7]. Since our approach models variables and memory contents explicitly, and it constructs intersection states in such a way that (memory) invariants are inferred and preserved, our approach is especially suitable for programs where termination depends on the contents of variables and addresses. However, a downside of this high precision is that it often takes long to construct symbolic execution graphs, since AProVE cannot give any meaningful answer before this construction is finished. The more information we try to keep in the abstract states, the more time is needed in every symbolic execution step when inferring knowledge for the next state. This results in a larger runtime than that of many other tools for termination analysis. Before developing the improvements suggested in this extended abstract, this used to result in many timeouts when analyzing large programs with many function calls, even if termination of the functions was not hard to prove once the graph was constructed. For every function call, an additional



(a) Programs of *Termination-Other* **(b)** Recursive programs of other subcategories

Figure 3 Number of termination proofs for leading tools in *SV-COMP*

subgraph of the SEG was computed in the non-modular approach of [12]. This did not only prohibit the handling of recursive functions but it also prevented an efficient treatment of programs with several calls of the same function.

Thus, the approach to analyze functions modularly is a big step towards scalability. To evaluate the power of the new approach, we use the results that AProVE and the other tools achieved at *SV-COMP*.[1] Fig. 3a shows the number of programs where termination was proved for the three leading tools of the *Termination* category of *SV-COMP* in AProVE's weakest subcategory *Termination-Other*, which was introduced in 2017. The bars in Fig. 3a indicate the total number of terminating programs. This subcategory mainly consists of large programs with significantly more function calls and branching instructions than there are in the programs of the remaining two subcategories. In 2017, AProVE already performed well on smaller recursive programs, but this approach was not yet generalized and optimized to use a modular analysis for non-recursive functions. In the following two years, AProVE substantially reduced the relative gap to the other leading tools for these kinds of examples.

Fig. 3b shows the number of recursive programs in the remaining two subcategories of *SV-COMP* where termination was proved. Here, we give the numbers of successful proofs for the three leading tools of the *Termination* category per year. Again, the bars indicate the total number of terminating recursive programs. Note that for most of the years, the set of programs is a proper superset of the set of programs of the previous year and the newly added programs tend to be harder to analyze. We see that first support to handle recursion was already very successfully implemented in the AProVE version of 2015. In the following years, this technique was further improved so that for most of the years, AProVE was able to prove termination for more of these programs than the other tools. Over the years, we repeatedly optimized and extended the techniques in AProVE to handle recursion and modularity, until we obtained the approach presented in this extended abstract in 2019.

The three leading tools of the *Termination* category of *SV-COMP* 2020 were UAutomizer, CPA-Seq, and 2LS. However, UAutomizer and CPA-Seq did not find more termination proofs for the programs in Fig. 3a and Fig. 3b than in 2019. 2LS was able to prove termination for

[1] *International Competition on Software Verification*, see `https://sv-comp.sosy-lab.org/`.

nearly as many programs as CPA-Seq in *Termination-Other*, but did not find any termination proofs for the recursive programs in other subcategories.

Apart from improving AProVE's capabilities for non-termination proofs, in future work we plan to extend our approach to handle recursive data structures. Here, the main challenge is to create heap invariants that reason about the shape of data structures and that abstract from their exact properties, but still contain sufficient knowledge about the memory contents needed for the termination proof. Similar to the approach sketched in the current extended abstract, this will require methods to remove and to restore knowledge about allocations in the abstract states in order to validate memory safety. Furthermore, these tasks have to combined with the handling of byte-precise pointer arithmetic.

References

1 M. Brockschmidt, C. Otto, and J. Giesl. Modular termination proofs of recursive Java Bytecode programs by term rewriting. In *Proc. RTA '11*, LIPIcs 10, pages 155–170, 2011. doi:10.4230/LIPIcs.RTA.2011.155.

2 M. Brockschmidt, B. Cook, S. Ishtiaq, H. Khlaaf, and N. Piterman. T2: Temporal property verification. In *Proc. TACAS '16*, LNCS 9636, pages 387–393, 2016. doi:10.1007/978-3-662-49674-9_22.

3 Clang. https://clang.llvm.org/.

4 L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. TACAS '08*, LNCS 4963, pages 337–340, 2008. doi:10.1007/978-3-540-78800-3_24.

5 B. Dutertre and L. de Moura. The Yices SMT solver, 2006. Tool paper at https://yices.csl.sri.com/papers/tool-paper.pdf.

6 N. Eén and N. Sörensson. An extensible SAT-solver. In *Proc. SAT '03*, LNCS 2919, pages 502–518, 2003. doi:10.1007/978-3-540-24605-3_37.

7 J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning*, 58(1):3–31, 2017. doi:10.1007/s10817-016-9388-y.

8 M. Haslbeck and R. Thiemann. An Isabelle/HOL formalization of AProVE's termination method for LLVM IR. In *Proc. CPP '21*, pages 238–249, 2021. doi:10.1145/3437992.3439935.

9 J. Hensel, F. Emrich, F. Frohn, T. Ströder, and J. Giesl. AProVE: Proving and disproving termination of memory-manipulating C programs (competition contribution). In *Proc. TACAS '17*, LNCS 10206, pages 350–354, 2017. doi:10.1007/978-3-662-54580-5_21.

10 J. Hensel, J. Giesl, F. Frohn, and T. Ströder. Termination and complexity analysis for programs with bitvector arithmetic by symbolic execution. *Journal of Logical and Algebraic Methods in Programming*, 97:105–130, 2018. doi:10.1016/j.jlamp.2018.02.004.

11 C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proc. CGO '04*, pages 75–88, 2004. doi:10.1109/CGO.2004.1281665.

12 T. Ströder, J. Giesl, M. Brockschmidt, F. Frohn, C. Fuhs, J. Hensel, P. Schneider-Kamp, and C. Aschermann. Automatically proving termination and memory safety for programs with pointer arithmetic. *Journal of Automated Reasoning*, 58(1):33–65, 2017. doi:10.1007/s10817-016-9389-x.

13 J. Zhao, S. Nagarakatte, M. M. K. Martin, and S. Zdancewic. Formalizing the LLVM IR for verified program transformations. In *Proc. POPL '12*, pages 427–440, 2012. doi:10.1145/2103621.2103709.

Loops for which Multiphase-Linear Ranking Functions are Sufficient

Amir M. Ben-Amram
School of Computer Science, The Tel-Aviv Academic College, Israel

Jesús J. Domenech
DSIC, Complutense University of Madrid (UCM), Spain

Samir Genaim
DSIC, Complutense University of Madrid (UCM), Spain

Abstract

In this paper we identify sub-classes of *single-path linear-constraint loops* loops for which *multiphase ranking functions* are sufficient, i.e., they terminate if and only if they have such ranking functions. This has some important consequences: (1) complete algorithms for such ranking functions can decide termination of these classes as well; and (2) terminating loops in these classes have linear run-time complexity.

2012 ACM Subject Classification Theory of computation → Program analysis

Keywords and phrases Ranking functions, Single-path linear constraint loops

Funding This work was funded partially by the Spanish MCIU, AEI and FEDER (EU) project RTI2018-094403-B-C31, by the CM project S2018/TCS-4314 co-funded by EIE Funds of the European Union, and by the UCM CT42/18-CT43/18 grant.

1 Introduction

In this paper, we are interested in termination analysis of *single-path linear-constraint loops* (*SLC* loops) using *multiphase ranking functions* (MΦRFs for short), in particular, in identifying sub-classes of *SLC* loops for which MΦRFs are sufficient, i.e., they terminate if and only if they have MΦRFs. This is important because, for such classes, a decision procedure for MΦRFs becomes a decision procedure for termination.

An *SLC* loop over n *rational* variables $x_1, \ldots, x_n$ has the form

$$\textit{while } (B\mathbf{x} \leq \mathbf{b}) \textit{ do } A\mathbf{x} + A'\mathbf{x}' \leq \mathbf{c} \tag{1}$$

where $\mathbf{x} = (x_1, \ldots, x_n)^{\mathrm{T}}$ and $\mathbf{x}' = (x'_1, \ldots, x'_n)^{\mathrm{T}}$ are column vectors, and for some $p, q > 0$, $B \in \mathbb{Q}^{p \times n}$, $A, A' \in \mathbb{Q}^{q \times n}$, $\mathbf{b} \in \mathbb{Q}^p$, $\mathbf{c} \in \mathbb{Q}^q$. The constraint $B\mathbf{x} \leq \mathbf{b}$ is called *the loop guard* and the other constraint is called *the update*. The update is *affine linear* if it can be rewritten as $\mathbf{x}' = U\mathbf{x} + \mathbf{c}$. We say that there is a transition from a state $\mathbf{x} \in \mathbb{Q}^n$ to a state $\mathbf{x}' \in \mathbb{Q}^n$, if $\mathbf{x}$ satisfies the loop condition and $\mathbf{x}$ and $\mathbf{x}'$ satisfy the update constraint. A transition can be seen as a point $\binom{\mathbf{x}}{\mathbf{x}'} \in \mathbb{Q}^{2n}$, where its first n components correspond to $\mathbf{x}$ and its last n components to $\mathbf{x}'$. We denote a transition $\binom{\mathbf{x}}{\mathbf{x}'}$ by $\mathbf{x}''$, and the set of all transitions by $\mathcal{Q}$ which is a polyhedron. The *projection* of $\mathcal{Q}$ onto the $\mathbf{x}$-space, i.e., the set of enabled states, is defined as $\mathtt{proj}_{\mathbf{x}}(\mathcal{Q}) = \{\mathbf{x} \in \mathbb{Q}^n \mid \exists \mathbf{x}'.\ \binom{\mathbf{x}}{\mathbf{x}'} \in \mathcal{Q}\}$. An *integer loop* is an *SLC* loop restricted to integer transitions (i.e., variables take only integer values). The following is an example of an *SLC* loop.

$$\texttt{while } (x_1 \geq -x_3) \texttt{ do } x'_1 = x_1 + x_2,\ x'_2 = x_2 + x_3,\ x'_3 = x_3 - 1 \tag{2}$$

The description of a loop may involve linear inequalities rather than equations, and consequently be non-deterministic.

Algorithm 1 A semi-decision procedure for existence of MΦRFs [1].

```
MLRF(Q)
begin
1   if (Q is empty) then return ∅
2   else
3       Compute the generators (a⃗_1, b_1), ..., (a⃗_l, b_l) of proj_x(Q)^#
4       Let Q' = Q ∧ a⃗_1 · x − a⃗_1 · x' ≤ 0 ∧ ··· ∧ a⃗_l · x − a⃗_l · x' ≤ 0
5       if (Q' == Q) then return Q
6       else return MLRF(Q')
```

Several kinds of ranking functions have been suggested for proving termination of *SLC* loops. In this paper, we are interested in *Multiphase ranking functions*. Intuitively, an MΦRF is a tuple $\langle f_1, \ldots, f_d \rangle$ of linear functions that define phases of the loop that are linearly ranked, as follows: f_1 decreases on all transitions, and when it becomes negative f_2 decreases, and when f_2 becomes negative, f_3 will decrease, and so on. Loop (2) has the MΦRF $\langle x_3 + 1, x_2 + 1, x_1 \rangle$. The parameter d is called the *depth* of the MΦRF.

The decision problem *Existence of an MΦRF* asks to determine whether an *SLC* loop has an MΦRF. The *bounded* decision problem restricts the search to MΦRFs of depth d, where d is part of the input. Ben-Amram and Genaim [3] showed that the bounded version of the MΦRF problem is PTIME for *SLC* loops with rational-valued variables, and coNP-complete for *SLC* loops with integer-valued variables. They also showed that, for *SLC* loops, MΦRFs have the same power as *lexicographic-linear ranking functions* and that they imply linear run-time complexity bounds. The problem of deciding if a given *SLC* admits an MΦRF, without a given bound on the depth, is still open, however, in a recent work [1] we suggested a semi-decision procedure that sheds some light on this class of ranking functions.

The procedure (see Algorithm 1) is based on using the set of non-negative functions[1] over the enabled states (Line 3) to continuously reduce $\mathcal{Q}$ (Line 4) until reaching an empty set of transitions (Line 1). If a loop $\mathcal{Q}$ has an MΦRF of optimal depth d, it is guaranteed that MLRF($\mathcal{Q}$) will reach an empty set in d recursive calls, and if the loop does not have an MΦRF it might find a recurrent set that witnesses non-termination (Line 5) or diverge.

In the rest of this paper, we demonstrate the usefulness of Algorithm 1 for studying properties of *SLC* loops, in particular, we use it to characterize kinds of *SLC* loops for which there is always an MΦRF, if the loop is terminating, and thus have linear run-time complexity. We shall prove this result for two kinds of loops, both considered in previous work, namely *octagonal relations* and *affine relations with the finite-monoid property* – for both classes, termination has been proven decidable [4]. We only consider the rational case.

2 Loops for which MΦRFs are sufficient

We let $\mathcal{Q}^n = \{\binom{\mathbf{x}}{\mathbf{z}} \mid \exists \mathbf{y} \,.\, \binom{\mathbf{x}}{\mathbf{y}} \in \mathcal{Q} \wedge \binom{\mathbf{y}}{\mathbf{z}} \in \mathcal{Q}^{n-1}\}$ where $\mathcal{Q}^0$ is the identity relation, i.e., $\mathcal{Q}^n$ is the n-th composition of $\mathcal{Q}$, which is a polyhedron. We let $\mathtt{pre}^n(\mathcal{Q}) = \mathtt{proj}_{\mathbf{x}}(\mathcal{Q}^n)$, which is the states from which we can make traces of length at least n (for non-deterministic loops

[1] The set of non-negative functions over $\mathcal{S} \subseteq \mathbb{Q}^n$ is $\mathcal{S}^{\#} = \{(\vec{a}, b) \in \mathbb{Q}^{n+1} \mid \forall \mathbf{x} \in \mathcal{S}.\ \vec{a} \cdot \mathbf{x} + b \geq 0\}$. When $\mathcal{S}$ is a polyhedron, $\mathcal{S}^{\#}$ is polyhedral cone, i.e., finitely generated by $(\vec{a}_1, b_1), \ldots, (\vec{a}_l, b_l)$.

some might be less than n as well). Our results rely on the following property of Algorithm 1.

▶ **Lemma 1** ([1]). *If $\mathcal{Q}'$ (on Line 4) has an MΦRF of optimal depth d, then $\mathcal{Q}$ has one of optimal depth $d+1$.*

2.1 Finite loops

First, we consider loops which always terminate and, moreover, their number of iterations is bounded by a constant, i.e., there is $N > 0$ s.t. $\mathcal{Q}^N = \emptyset$. Note that such a loop terminates in at most $N-1$ iterations ($N-1$ is an upper-bound on the length of its traces).

▶ **Lemma 2.** *If $\mathcal{Q}^N = \emptyset$, then it has an MΦRF of depth less than N.*

Proof. The proof is by induction on N. For $N = 1$, $\mathcal{Q} = \emptyset$, and it has an MΦRF of depth zero. Let $N > 1$, and assume that $\mathcal{Q}^{N-1} \neq \emptyset$, otherwise it trivially follows for N. Consider a transition $\mathbf{x}'' = \binom{\mathbf{x}}{\mathbf{x}'}$ that is the last in a terminating trace. We have $\mathbf{x} \in \mathtt{proj}_{\mathbf{x}}(\mathcal{Q})$ and $\mathbf{x}' \notin \mathtt{proj}_{\mathbf{x}}(\mathcal{Q})$. Since $\mathtt{proj}_{\mathbf{x}}(\mathcal{Q})$ is a closed polyhedral set, this means that there is a function ρ, defined by some $(\vec{a}, b) \in \mathtt{proj}_{\mathbf{x}}(\mathcal{Q})^{\#}$, that is non-negative over $\mathtt{proj}_{\mathbf{x}}(\mathcal{Q})$ but negative on $\mathbf{x}'$, and thus $\rho(\mathbf{x}) - \rho(\mathbf{x}') > 0$. It follows that $\mathbf{x}''$ is eliminated by Algorithm 1 when computing $\mathcal{Q}'$ on Line 4. This means that any transition of $\mathcal{Q}'$ cannot be the last transition of any terminating run of $\mathcal{Q}$, and thus $(\mathcal{Q}')^{N-1} = \emptyset$. Therefore, by induction, it has an MΦRF of depth at most $N-2$, and by Lemma 1, $\mathcal{Q}$ has an MΦRF of depth at most $N-1$. ◀

2.2 The class $\mathtt{RF}(b)$

This class contains *terminating* loops which can be described as having the following behavior: Transitions are linearly ranked, as long as we are in states from which we can make runs of length at least b. In other words, once we reach a state from which we cannot make more than $b-1$ transitions we do not require the rest of the trace to be linearly ranked.

▶ **Definition 3.** *We say that an SLC loop $\mathcal{Q}$ belongs to the class $\mathtt{RF}(b)$ if the loop $\mathcal{Q} \cap \{\binom{\mathbf{x}}{\mathbf{x}'} \in \mathbb{Q}^{2n} \mid \mathbf{x} \in \mathtt{pre}^b(\mathcal{Q})\}$ has a linear ranking function (LRF for short).*

We note that $\mathtt{RF}(1)$ is the class of loops which have a *linear ranking function*.

▶ **Lemma 4.** *Loops in $\mathtt{RF}(b)$ have MΦRFs of depth at most b.*

Proof. This lemma generalizes Lemma 2, since the loops concerned there are $\mathtt{RF}(N-1)$. The proof is done similarly by induction on b. For $b = 1$, $\mathcal{Q}$ has a *LRF* by definition. Let $b > 1$, and suppose that $\mathbf{x}'' = \binom{\mathbf{x}}{\mathbf{x}'} \in \mathcal{Q}$ is a last transition of a terminating run, then $\mathbf{x} \in \mathtt{proj}_{\mathbf{x}}(\mathcal{Q})$ and $\mathbf{x}' \notin \mathtt{proj}_{\mathbf{x}}(\mathcal{Q})$. Since $\mathtt{proj}_{\mathbf{x}}(\mathcal{Q})$ is a closed polyhedral set, this means that there is a function ρ, defined by some $(\vec{a}, b) \in \mathtt{proj}_{\mathbf{x}}(\mathcal{Q})^{\#}$, that is non-negative over $\mathtt{proj}_{\mathbf{x}}(\mathcal{Q})$ but negative over $\mathbf{x}'$, and thus $\rho(\mathbf{x}) - \rho(\mathbf{x}') > 0$. It follows that $\mathbf{x}''$ is eliminated by Algorithm 1 when computing $\mathcal{Q}'$ on Line 4. This means that any transition of $\mathcal{Q}'$ cannot be the last transition of any terminating run of $\mathcal{Q}$, and thus $\mathcal{Q}'$ is $\mathtt{RF}(b-1)$. Therefore, by induction, it has an MΦRF of depth at most $b-1$, and by Lemma 1 $\mathcal{Q}$ has an MΦRF of depth at most b. ◀

▶ **Example 5.** Consider the loop [4] defined by

$$\mathcal{Q} = \{x_2 - x_1' \leq -1,\ x_3 - x_2' \leq 0,\ x_1 - x_3' \leq 0,\ x_4' - x_4 \leq 0,\ x_3' - x_4 \leq 0\}. \tag{3}$$

This loop is $\mathtt{RF}(3)$, since adding $\mathtt{pre}^3(Q) = \{x_2 + x_4 \geq 1,\ x_3 + x_4 \geq 1,\ x_1 + x_4 \geq 0\}$ to the loop we find a *LRF*, e.g., $\rho(\mathbf{x}) = -x_1 - x_2 - x_3 + 3x_4 + 1$. Indeed, $\mathcal{Q}$ has an MΦRF of

optimal depth 3, e.g., $\langle -x_1 - x_2 - x_3 + 3x_4 + 1, -\frac{2}{3}x_1 - \frac{1}{3}x_2 + x_4 + 1, -\frac{1}{4}x_1 + \frac{1}{4}x_4 + 1\rangle$. Note that the first component is the *LRF* that we have found for $\mathcal{Q} \cap \{\mathbf{x}'' \mid \mathbf{x} \in \mathtt{pre}^3(Q)\}$. □

Note that if we know that a given class of loops belongs to $\mathtt{RF}(b)$, then bounding the recursion depth of Algorithm 1 by b gives us a decision procedure for the existence of an MΦRF for this class. Bozga, Iosif and Konecný [4] proved that *octagonal relations*[2] are $\mathtt{RF}(5^{2n})$, where n is the number of variables. Thus for octagonal relations, we can decide termination and for terminating loops obtain MΦRFs. For the depth of the MΦRF, namely the parameter b above, Bozga, Iosif and Konecný [4] gives a tighter (polynomial) result for those octagonal relations which allow arbitrarily long executions (called $*$-consistent).

2.3 Loops with affine-linear updates

In certain cases, we can handle loops with affine-linear updates – which are, in general, not octagonal. Recall that a loop with affine-linear update has a transition relation of the form:

$$\mathcal{Q} \equiv [B\mathbf{x} \leq \mathbf{b} \wedge \mathbf{x}' = U\mathbf{x} + \mathbf{c}]\,. \tag{4}$$

We keep the meaning of the symbols $U, B, \mathbf{b}, \mathbf{c}$ fixed for the sequel. Moreover, we express the loop using the transformation $\mathcal{U}(\mathbf{x}) = U\mathbf{x} + \mathbf{c}$ and the guard $\mathcal{G} \equiv [B\mathbf{x} \leq \mathbf{b}]$. We use U_{ij} to denote the entry of matrix U in row i and column j, and for a vector $\mathbf{v}$ we let $\mathbf{v}[i..j]$ be the vector obtained from components i to j of the vector $\mathbf{v}$.

Our goal is to show that if U^p, for some $p > 0$, is diagonalizable and all its eigenvalues are in $\{0, 1\}$, then $\mathcal{Q}$ is $\mathtt{RF}(3p)$, and thus, by Lemma 4, if terminating, it has an MΦRF. Affine loops with the *finite monoid property* that has been addressed by Bozga, Iosif and Konecný [4], satisfy this condition. Moreover, the existence of p, given the matrix U, is decidable [4, Section 5.3] so in principle we can decide if it exists and if it does,search for it by brute force.

We state some auxiliary lemmas first.

▶ **Lemma 6.** *Let $\mathcal{Q}$ be an affine-linear loop as in (4) such that for some $N > 0$, $\mathcal{Q}^N$ is $\mathtt{RF}(b)$. Then $\mathcal{Q}$ is $\mathtt{RF}(N(b+1))$.*

Proof. If $\mathcal{Q}^N$ is $\mathtt{RF}(b)$, then $\mathcal{Q}^N \cap \{\mathbf{x}'' \mid \mathbf{x} \in \mathtt{pre}^b(\mathcal{Q}^N)\}$ has a *LRF* ρ, and thus

$$\mathbf{x} \in \mathtt{pre}^b(\mathcal{Q}^N) = \mathtt{pre}^{Nb}(\mathcal{Q}) \;\Rightarrow\; \rho(\mathbf{x}) \geq 0 \,\wedge\, \rho(\mathbf{x}) - \rho(\mathcal{U}^N(\mathbf{x})) > 0\,. \tag{5}$$

Note that $\rho(\mathbf{x}) - \rho(\mathcal{U}^N(\mathbf{x}))$ can be written as

$$\sum_{j=0}^{N-1} \rho(\mathcal{U}^j(\mathbf{x})) - \sum_{j=0}^{N-1} \rho(\mathcal{U}^{j+1}(\mathbf{x})) \tag{6}$$

This is because every term $\rho(\mathcal{U}^i(\mathbf{x}))$, except for $i = 0$ and $i = N$, appear in (6) with positive and negative signs. Hence, if we let $\rho_1(\mathbf{x}) = \sum_{j=0}^{N-1} \rho(\mathcal{U}^j(\mathbf{x}))$ then:

$$\mathbf{x} \in \mathtt{pre}^{Nb}(\mathcal{Q}) \;\Rightarrow\; \rho_1(\mathbf{x}) - \rho_1(\mathcal{U}(\mathbf{x})) > 0\,. \tag{7}$$

Moreover, ρ_1 is the sum of terms $\rho(\mathcal{U}^i(\mathbf{x}))$ which are bounded from below on $\mathtt{pre}^{N(b+1)}(\mathcal{Q})$. Hence, we have a *LRF* for $\mathcal{Q} \cap \{\mathbf{x}'' \mid \mathbf{x} \in \mathtt{pre}^{N(b+1)}(\mathcal{Q})\}$ and thus $\mathcal{Q}$ is $\mathtt{RF}(N(b+1))$. ◀

[2] Conjunction of inequalities of the form $ax + by \leq c$ where $a, b \in \{-1, 0, 1\} and c \in \mathbb{Q}$.

▶ **Lemma 7.** *Let $\mathcal{Q}$ be a loop as in* (4)*, and assume U is diagonal with entries in* $\{0,1\}$*. Then, if $\mathcal{Q}$ is terminating, it is* $\mathtt{RF}(2)$*.*

Proof. W.l.o.g. we may assume that $U_{11} = \cdots = U_{kk} = 1$ and $U_{jj} = 0$ for $j > k$, otherwise we could reorder the variables to put it into this form. Clearly, the update adds $\mathbf{c}_1 = \mathbf{c}[1..k]$ to the first k elements of $\mathbf{x}$, and sets the rest to $\mathbf{c}_2 = \mathbf{c}[k+1..n]$. Consequently, such a loop is non-terminating iff the space $V = \{\mathbf{x} \in \mathbb{Q}^n \mid \mathbf{x}[k+1..n] = \mathbf{c}_2\}$ intersects the loop guard $\mathcal{G} \equiv [B\mathbf{x} \leq \mathbf{b}]$, and $\mathbf{u} = (c_1, \dots c_k, 0, \dots, 0)^{\mathrm{T}} \in \mathbb{Q}^n$ is a recession direction of the guard, i.e., $B\mathbf{u} \leq \mathbf{0}$. To see this: suppose these conditions hold, then starting from any state $\mathbf{x}_0 \in V \cap \mathcal{G}$, the state after i iterations will be $\mathbf{x}_i = \mathbf{x}_0 + i\mathbf{u}$, which is in $\mathcal{G}$ since $\mathbf{x}_0 \in \mathcal{G}$ and $\mathbf{u}$ is a recession direction of $\mathcal{G}$, and thus the execution does not terminate; for the other direction, suppose it does not terminate, then there must be a non-terminating execution that starts in $\mathbf{x}_0 \in V$, this execution generates the states $\mathbf{x}_0 + i\mathbf{u} \in \mathcal{G}$ and thus $\mathbf{u}$ is a recession direction of $\mathcal{G}$.

Now suppose the loop is terminating, we show that it is $\mathtt{RF}(2)$. Let us analyze a run of the loop starting with some valid transition $\binom{\mathbf{x}_0}{\mathbf{x}_1}$. We have two cases:

1. If $\mathbf{x}_1 \notin \mathcal{G}$, then the run terminates in 1 iteration;
2. If $\mathbf{x}_1 \in \mathcal{G}$, then V intersects with $\mathcal{G}$, since $\mathbf{x}_1[k+1..n] = \mathbf{c}_2$, and thus $B\mathbf{u} \leq \mathbf{0}$ should not hold, otherwise the loop is non-terminating. This means that there is a constraint $\vec{b} \cdot \mathbf{x} \leq b$ of the guard such that $\vec{b} \cdot \mathbf{u} > 0$. Define $\rho(\mathbf{x}) = -\vec{b} \cdot \mathbf{x} + b$, and note that it is non-negative on all states in the run except the last (which is not in the guard). Write the initial state $\mathbf{x}_0$ as $\binom{\mathbf{x}_0[1..k]}{\mathbf{x}_0[k+1..n]}$, and note that the i-th state, for $i \geq 1$, is $\mathbf{x}_i = \binom{\mathbf{x}_0[1..k]+i\mathbf{c_1}}{\mathbf{c_2}}$. Then, for $i \geq 1$, we have $\rho(\mathbf{x}_i) - \rho(\mathbf{x}_{i+1}) = \vec{b} \cdot \binom{\mathbf{c}_1}{\mathbf{0}} = \vec{b} \cdot \mathbf{u} > 0$ which means that ρ is a *LRF* from the second transition on (it is not guaranteed that $\rho(\mathbf{x}_0) - \rho(\mathbf{x}_1) > 0$). Now take $\rho'(\mathbf{x}) = \rho(\mathcal{U}(\mathbf{x})) = \vec{a} \cdot U\mathbf{x} + \vec{a} \cdot \mathbf{c} + b$, and note that it is a *LRF* until the transition before the last – because ρ' looks one state ahead, by considering $\mathcal{U}(\mathbf{x})$ instead of $\mathbf{x}$, so unlike ρ it is decreasing on the first transition as well but might be negative on the last one.

This analysis implies that any terminating trace is either of length 1, or has a *LRF* until one transition before the last, that is $\mathtt{RF}(2)$. ◀

Now we are in a position for proving our main result of this section.

▶ **Lemma 8.** *If U^p, for some $p > 0$, is diagonalizable and all its eigenvalues are in* $\{0,1\}$*, then loop* (4) *is either non-terminating or* $\mathtt{RF}(3p)$*.*

Proof. Recall that the update is $\mathcal{U}(\mathbf{x}) = U\mathbf{x} + \mathbf{c}$, then $\mathcal{U}^p(\mathbf{x}) = U^p\mathbf{x} + \mathbf{v}$, for a vector $\mathbf{v} = (I + U + \cdots + U^{p-1})\mathbf{c}$. Taking into account the guard,

$$\mathcal{Q}^p \equiv (B\mathbf{x} \leq \mathbf{b} \wedge \cdots \wedge B\mathcal{U}^{p-1}(\mathbf{x}) \leq \mathbf{b}) \wedge \mathbf{x}' = \mathcal{U}^p(\mathbf{x})\,. \tag{8}$$

We write this guard concisely with the notation $B^{\langle p \rangle}\mathbf{x} \leq \mathbf{b}^{\langle p \rangle}$. Since, by assumption, U^p is diagonalizable, there is a non-singular matrix P and a diagonal matrix D such that $P^{-1}U^pP = D$ and D has only 1's and 0's on the diagonal (P is a change-of-basis transformation). We consider a loop $\widehat{\mathcal{Q}^p}$ similar to $\mathcal{Q}^p$ but transformed by P, that is:

$$\widehat{\mathcal{Q}^p} \equiv B^{\langle p \rangle}P\mathbf{x} \leq \mathbf{b}^{\langle p \rangle} \wedge \mathbf{x}' = D\mathbf{x} + P^{-1}\mathbf{v}\,. \tag{9}$$

Properties like termination and linear ranking are not affected by such a change of basis.

This is because if $\binom{\mathbf{x}_0}{\mathbf{x}_1}$ is a transition of $\mathcal{Q}^p$ then $\binom{P^{-1}\mathbf{x}_0}{P^{-1}\mathbf{x}_1}$ is a transition of $\widehat{\mathcal{Q}^p}$, and if $\binom{\mathbf{x}_0}{\mathbf{x}_1}$ is a transition of $\widehat{\mathcal{Q}^p}$ then $\binom{P\mathbf{x}_0}{P\mathbf{x}_1}$ is a transition of $\mathcal{Q}^p$. This means that there is a one-to-one correspondence between the traces. Moreover, if function $\vec{a} \cdot \mathbf{x} + b$ ranks a

transition of $\mathcal{Q}^p$ then $(\vec{a}P^{-1}) \cdot \mathbf{x} + b$ ranks the corresponding transition of $\widehat{\mathcal{Q}^p}$, and if it ranks a transition $\widehat{\mathcal{Q}^p}$ then $(\vec{a}P) \cdot \mathbf{x} + b$ ranks the corresponding transition of $\mathcal{Q}^p$. We conclude that, if terminating, $\mathcal{Q}^p$ is $\mathtt{RF}(b)$ iff $\widehat{\mathcal{Q}^p}$ is $\mathtt{RF}(b)$.

Now, $\widehat{\mathcal{Q}^p}$ has the diagonal form discussed in Lemma 7, and thus, in the case that it terminates, it is $\mathtt{RF}(2)$ and so is $\mathcal{Q}^p$. Then using Lemma 6 we conclude that $\mathcal{Q}$ is $\mathtt{RF}(3p)$. ◀

3 Conclusions

We have shown that loops defined by *octagonal relations* and *affine relations with the finite-monoid property*, terminate iff they have MΦRFs. This means that complete algorithms for MΦRFs can decide termination of these classes as well, and that terminating loops in these classes have linear run-time complexity. Another question, which we did not address and leave for future work, is whether we can ensure that Algorithm 1 recognizes the non-terminating members of these classes, i.e., whether Algorithm 1 (without a bound on the depth) would always stop for all programs from these classes.

In previous sections, we have restricted ourself to loops over the rationals, however, all results hold for loop over the reals as well. Besides, since a loop over the integers has an MΦRF if and only if its integer-hull has an MΦRF over the rational [3], our results are valid for the integer case if integer-hull of the loop falls in one of the classes that we discussed in this paper. Moreover, it is know that for affine loops as (4) in which all numbers in U and $\mathbf{c}$ are integer, computing the integer-hull can be done by computing the integer-hull of the loop condition [2]. This means that, for such loops, the result of Section 2.3 are still valid for the integer case as well.

References

1 Amir M. Ben-Amram, Jesús J. Doménech, and Samir Genaim. Multiphase-linear ranking functions and their relation to recurrent sets. In Bor-Yuh Evan Chang, editor, *Proceedings of the 26th International Symposium on Static Analysis (SAS'19)*, volume 11822 of *Lecture Notes in Computer Science*, pages 459–480. Springer, 2019.

2 Amir M. Ben-Amram and Samir Genaim. Ranking functions for linear-constraint loops. *Journal of the ACM*, 61(4):26:1–26:55, July 2014.

3 Amir M. Ben-Amram and Samir Genaim. On multiphase-linear ranking functions. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification, CAV'17*, volume 10427 of *Lecture Notes in Computer Science*, pages 601–620. Springer, 2017.

4 Marius Bozga, Radu Iosif, and Filip Konecný. Deciding conditional termination. *Logical Methods of Computer Science*, 10(3), 2014.

Analyzing Expected Runtimes of Probabilistic Integer Programs Using Expected Sizes

Fabian Meyer
LuFG Informatik 2, RWTH Aachen University, Aachen, Germany

Marcel Hark
LuFG Informatik 2, RWTH Aachen University, Aachen, Germany

Jürgen Giesl
LuFG Informatik 2, RWTH Aachen University, Aachen, Germany

Abstract

We present a novel modular approach to infer upper bounds on the expected runtimes of probabilistic integer programs automatically. To this end, it computes bounds on the runtimes of program parts and on the sizes of their variables in an alternating way. To evaluate its power, we implemented our approach in a new version of our open-source tool KoAT.

2012 ACM Subject Classification Theory of computation → Program analysis; Theory of computation → Random walks and Markov chains; Mathematics of computing → Probabilistic algorithms

Keywords and phrases Probabilistic Integer Programs, Expected Runtimes, Expected Sizes, Automatic Complexity Analysis, Positive Almost Sure Termination

Funding funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 235950644 (Project GI 274/6-2) and the DFG Research Training Group 2236 UnRAVeL

Acknowledgements We thank Carsten Fuhs for discussions on initial ideas.

1 Introduction

Most approaches for automatic complexity analysis combine basic techniques like *ranking functions* in sophisticated ways. For example, in [3] we developed a modular approach for complexity analysis of integer programs, based on alternating between finding runtime bounds for program parts and using them to infer bounds on the sizes of variables in such parts. The corresponding implementation in KoAT is one of the leading tools for complexity analysis.

There are several adaptions of basic techniques like ranking functions, but most sophisticated full approaches for complexity analysis have not been adapted to *probabilistic programs* yet. In this paper, we study probabilistic integer programs and define suitable notions of non-probabilistic and expected runtime and size bounds (Sect. 2). Then, we adapt our modular approach for runtime and size analysis of [3] to probabilistic programs (Sect. 3). So such an adaption is not only possible for *basic techniques* like ranking functions, but also for *full approaches* for complexity analysis. When computing expected runtime or size bounds for new program parts, a main difficulty is to determine when it is sound to use *expected* bounds on previous program parts and when one has to use *non-probabilistic* bounds instead. Sect. 4 evaluates the implementation of our new approach in KoAT [3, 4]. The full version of our paper appeared in [5].

2 Complexity Bounds

Fig. 1 shows a *probabilistic integer program* (PIP), with the *locations* $\mathcal{L} = \{\ell_0, \ell_1, \ell_2\}$ and the variables $\mathcal{V} = \{x, y\}$. It has four *general transitions* $\mathcal{GT} = \{g_0, g_1, g_2, g_3\}$ with $g_0 = \{t_0\}$, $g_1 = \{t_1, t_2\}$, $g_2 = \{t_3\}$, and $g_3 = \{t_4\}$, where the *transitions* t_i are chosen with a certain

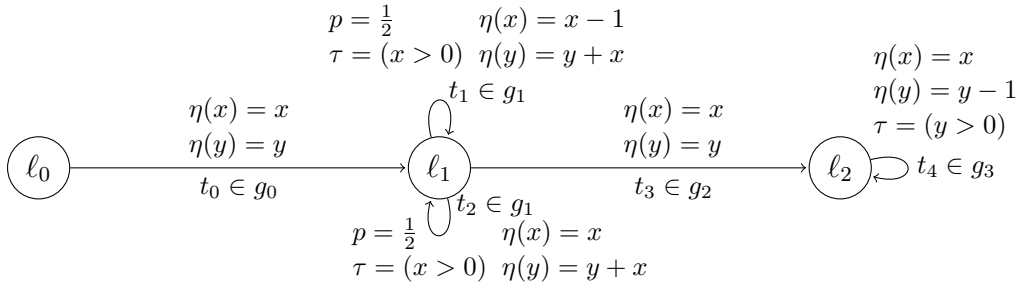


Figure 1 PIP with non-deterministic branching (g_1 vs. g_2) and probabilistic branching (t_1 vs. t_2)

probability p when their general transition is executed. Transitions have a *guard* τ and perform an *update* η on the program variables. Let $p = 1$ and $\tau = \mathbf{t}$ if not stated explicitly.

Similar to [3], our approach computes bounds that represent *weakly monotonically increasing* functions from $\mathcal{V} \to \mathbb{R}_{\geq 0} \cup \{\infty\}$. Such bounds can easily be "composed", i.e., if f and g are both weakly monotonically increasing upper bounds, then so is $f \circ g$.

For the set of all transitions $\mathcal{T}$ and the set of all bounds $\mathcal{B}$, we call $\mathcal{RB}\colon \mathcal{T} \to \mathcal{B}$ a *runtime bound* if for all $t \in \mathcal{T}$, $\mathcal{RB}(t)$ is an upper bound on the number of executions of t. $\mathcal{SB}\colon \mathcal{T} \times \mathcal{V} \to \mathcal{B}$ is a *size bound* if for all $t \in \mathcal{T}$ and $x \in \mathcal{V}$, $\mathcal{SB}(t, x)$ over-approximates the greatest absolute value that x takes after the application of t. We call a tuple $(\mathcal{RB}, \mathcal{SB})$ a (non-probabilistic) *bound pair*. We use such non-probabilistic bound pairs for an initialization of expected bounds and to compute improved expected runtime and size bounds in Sect. 3.

▶ **Example 1** (Bound Pair). *The technique of [3] computes the following bound pair for the PIP of Fig. 1 (by ignoring the probabilities of the transitions).*

$$\mathcal{RB}(t) = \begin{cases} 1, & \text{if } t = t_0 \text{ or } t = t_3 \\ x, & \text{if } t = t_1 \\ \infty, & \text{if } t = t_2 \text{ or } t = t_4 \end{cases} \qquad \mathcal{SB}(t, x) = \begin{cases} x, & \text{if } t \in \{t_0, t_1, t_2\} \\ 3 \cdot x, & \text{if } t \in \{t_3, t_4\} \end{cases}$$

$$\mathcal{SB}(t, y) = \begin{cases} y, & \text{if } t = t_0 \\ \infty, & \text{if } t \in \{t_1, t_2, t_3, t_4\} \end{cases}$$

Thus, the runtimes of t_2 and t_4 are unbounded (i.e., the PIP is not terminating when regarding it as a non-probabilistic program). $\mathcal{SB}(t, x)$ is finite for all transitions t, since x is never increased.[1] *In contrast, the value of y can be arbitrarily large after all transitions but t_0.*

We now define the *expected* runtime and size complexity of a PIP $\mathcal{P}$. For a general transition $g \in \mathcal{GT}$, its *runtime* is the random variable $\mathcal{R}(g)$, where for any run ϑ (i.e., for any infinite sequence of configurations of the program), $\mathcal{R}(g)(\vartheta)$ is the number of executions of a transition from g in the run ϑ. One can define the semantics of PIPs by a standard cylinder construction based on deterministic Markovian schedulers. Then for any scheduler $\mathfrak{S}$ and any initial state s_0 mapping the variables to integers, one can define the *expected runtime complexity* of g to be the expected value $\mathbb{E}_{\mathfrak{S},s_0}(\mathcal{R}(g))$ of $\mathcal{R}(g)$ under the corresponding probability measure. The expected runtime complexity of the whole program $\mathcal{P}$ results from adding the expected runtime complexities of all its general transitions. If $\mathcal{P}$'s expected runtime complexity is finite for every scheduler $\mathfrak{S}$ and every initial state s_0, then $\mathcal{P}$ is called *positively almost surely terminating (PAST)* [2].

[1] The reason for $\mathcal{SB}(t_3, x) = 3 \cdot x$ is that this size bound should be the maximum of the size bounds $\mathcal{SB}(t, x)$ for all transitions t that may precede t_3, and we over-approximate the maximum of bounds by their sum. Therefore, we obtain $\mathcal{SB}(t_3, x) = \mathcal{SB}(t_0, x) + \mathcal{SB}(t_1, x) + \mathcal{SB}(t_2, x)$.

Similarly, for any $g \in \mathcal{GT}$, $\ell \in \mathcal{L}$, and $x \in \mathcal{V}$, their *size* is the random variable $\mathcal{S}(g,\ell,x)$, where for any run ϑ, $\mathcal{S}(g,\ell,x)(\vartheta)$ is the greatest absolute value of x in location ℓ, whenever ℓ was entered with a transition from g. For any scheduler $\mathfrak{S}$ and initial state s_0, the *expected size complexity* of (g,ℓ,x) is $\mathbb{E}_{\mathfrak{S},s_0}(\mathcal{S}(g,\ell,x))$. Our goal is to compute bounds $\mathcal{RB}_{\mathbb{E}}$ and $\mathcal{SB}_{\mathbb{E}}$ for the expected runtime and size complexity which hold *independent* of the scheduler.

▶ **Definition 2** (Expected Runtime and Size Bounds). *$\mathcal{RB}_{\mathbb{E}} : \mathcal{GT} \to \mathcal{B}$ is an* expected runtime bound *if $|s_0|\,(\mathcal{RB}_{\mathbb{E}}(g)) \geq \mathbb{E}_{\mathfrak{S},s_0}(\mathcal{R}(g))$ holds for all $g \in \mathcal{GT}$, schedulers $\mathfrak{S}$, and initial states s_0. Here, $|s_0|\,(\mathcal{RB}_{\mathbb{E}}(g))$ results from $\mathcal{RB}_{\mathbb{E}}(g)$ by instantiating every variable x with $|s_0(x)|$.*

$\mathcal{SB}_{\mathbb{E}} : \mathcal{GT} \times \mathcal{L} \times \mathcal{V} \to \mathcal{B}$ is an expected size bound *if $|s_0|\,(\mathcal{SB}_{\mathbb{E}}(g,\ell,x)) \geq \mathbb{E}_{\mathfrak{S},s_0}(\mathcal{S}(g,\ell,x))$ holds for all $g \in \mathcal{GT}$, $\ell \in \mathcal{L}$, $x \in \mathcal{V}$, and all schedulers $\mathfrak{S}$ and initial states s_0. A pair $(\mathcal{RB}_{\mathbb{E}}, \mathcal{SB}_{\mathbb{E}})$ is called an* expected bound pair.

▶ **Example 3** (Expected Runtime and Size Bounds). *Our new technique from Sect. 3 will derive the following expected bounds for the PIP from Fig. 1.*

$$\mathcal{RB}_{\mathbb{E}}(g) = \begin{cases} 1, & \text{if } g \in \{g_0, g_2\} \\ 2 \cdot x, & \text{if } g = g_1 \\ 6 \cdot x^2 + 2 \cdot y, & \text{if } g = g_3 \end{cases} \qquad \mathcal{SB}_{\mathbb{E}}(g, \_, x) = \begin{cases} x, & \text{if } g = g_0 \\ 2 \cdot x, & \text{if } g = g_1 \\ 3 \cdot x, & \text{if } g \in \{g_2, g_3\} \end{cases}$$

$$\mathcal{SB}_{\mathbb{E}}(g_0, \ell_1, y) = y \qquad \mathcal{SB}_{\mathbb{E}}(g_2, \ell_2, y) = 6 \cdot x^2 + 2 \cdot y$$

$$\mathcal{SB}_{\mathbb{E}}(g_1, \ell_1, y) = 6 \cdot x^2 + y \qquad \mathcal{SB}_{\mathbb{E}}(g_3, \ell_2, y) = 12 \cdot x^2 + 4 \cdot y$$

While the runtimes of t_2 and t_4 were unbounded in the non-probabilistic case (Ex. 1), we obtain finite bounds on the expected runtimes of $g_1 = \{t_1, t_2\}$ and $g_3 = \{t_4\}$. For example, we can expect x to be non-positive after at most $|s_0|\,(2 \cdot x)$ iterations of g_1. Based on the above expected runtime bounds, the expected runtime complexity of the PIP is at most $|s_0|\,(\mathcal{RB}_{\mathbb{E}}(g_0) + \ldots + \mathcal{RB}_{\mathbb{E}}(g_3)) = |s_0|\,(2 + 2 \cdot x + 2 \cdot y + 6 \cdot x^2)$, i.e., it is in $\mathcal{O}(n^2)$ where n is the maximal absolute value of the program variables at the start of the program.

3 Computing Expected Runtime Bounds

We use a class of probabilistic ranking functions $\mathfrak{r}$ that map every location to a *linear polynomial*. Nevertheless, our approach of course also infers non-linear expected runtimes (by combining the linear bounds obtained for different program parts). For any subsets $\mathcal{GT}_{>} \subseteq \mathcal{GT}_{\text{ni}} \subseteq \mathcal{GT}$, we then say that $\mathfrak{r}$ is a *probabilistic linear ranking function (PLRF)* for $\mathcal{GT}_{>}$ and $\mathcal{GT}_{\text{ni}}$ if all general transitions $g \in \mathcal{GT}_{\text{ni}}$ are non-increasing (i.e., $s(\tau_g) = \mathbf{t}$ implies $s(\mathfrak{r}(\ell_g)) \geq \exp_{\mathfrak{r},g,s}$) and all general transitions $g \in \mathcal{GT}_{>}$ are decreasing (i.e., $s(\tau_g) = \mathbf{t}$ implies $s(\mathfrak{r}(\ell_g)) - 1 \geq \exp_{\mathfrak{r},g,s}$). Here, τ_g is the condition and ℓ_g is the start location of g, and $\exp_{\mathfrak{r},g,s}$ denotes the expected value of $\mathfrak{r}$ after an execution of g in state s. Moreover, we need appropriate boundedness conditions concerning the positivity of the ranking function.

So if one is restricted to the sub-program $\mathcal{GT}_{\text{ni}}$, then $\mathfrak{r}(\ell)$ is an upper bound on the expected number of applications of transitions from $\mathcal{GT}_{>}$ when starting in ℓ. Hence, a PLRF for $\mathcal{GT}_{>} = \mathcal{GT}_{\text{ni}} = \mathcal{GT}$ would imply that the whole program is PAST. However, our PLRFs differ from the standard notion of probabilistic ranking functions by considering arbitrary subsets $\mathcal{GT}_{\text{ni}} \subseteq \mathcal{GT}$. This is needed for the modularity of our approach which allows us to analyze program parts separately (e.g., $\mathcal{GT} \setminus \mathcal{GT}_{\text{ni}}$ is ignored when inferring a PLRF).

▶ **Example 4** (PLRFs). *Consider again the PIP in Fig. 1 and the sets $\mathcal{GT}_{>} = \mathcal{GT}_{\text{ni}} = \{g_1\}$ and $\mathcal{GT}'_{>} = \mathcal{GT}'_{\text{ni}} = \{g_3\}$, which correspond to its two loops. The function $\mathfrak{r}$ with $\mathfrak{r}(\ell_1) = 2 \cdot x$*

and $\mathfrak{r}(\ell_0) = \mathfrak{r}(\ell_2) = 0$ *is a PLRF for* $\mathcal{GT}_> = \mathcal{GT}_{\text{ni}}$*: For* $s_1(x) = s(x-1) = s(x) - 1$ *and* $s_2(x) = s(x)$ *we have* $\exp_{\mathfrak{r},g,s} = \frac{1}{2} \cdot s_1(\mathfrak{r}(\ell_1)) + \frac{1}{2} \cdot s_2(\mathfrak{r}(\ell_1)) = 2 \cdot s(x) - 1 = s(\mathfrak{r}(\ell_1)) - 1$*. So* $\mathfrak{r}$ *is decreasing on* g_1 *and as* $\mathcal{GT}_> = \mathcal{GT}_{\text{ni}}$*, also the non-increase property holds.*
Similarly, $\mathfrak{r}'$ *with* $\mathfrak{r}'(\ell_2) = y$ *and* $\mathfrak{r}'(\ell_0) = \mathfrak{r}'(\ell_1) = 0$ *is a PLRF for* $\mathcal{GT}'_> = \mathcal{GT}'_{\text{ni}}$*.*

Our approach to infer expected runtime bounds is based on an underlying (non-probabilistic) bound pair $(\mathcal{RB}, \mathcal{SB})$ which is computed by existing techniques (in our implementation, we use [3]). To do so, we abstract the PIP to a standard integer transition system. Of course, we usually have $\mathcal{RB}(t) = \infty$ for some transitions t.

We start with an expected bound pair $(\mathcal{RB}_{\mathbb{E}}, \mathcal{SB}_{\mathbb{E}})$ that is obtained from "lifting" $(\mathcal{RB}, \mathcal{SB})$ to expected bounds by simply adding the bounds for all transitions in a general transition.

Afterwards, the expected runtime bound $\mathcal{RB}_{\mathbb{E}}$ is improved repeatedly by applying the following Thm. 6 (and $\mathcal{SB}_{\mathbb{E}}$ is improved repeatedly in a similar way). Here, we only show the improvement of $\mathcal{RB}_{\mathbb{E}}$ and refer to [5] for the improvement of expected size bounds. Our approach alternates the improvement of $\mathcal{RB}_{\mathbb{E}}$ and $\mathcal{SB}_{\mathbb{E}}$, and it uses expected size bounds on "previous" transitions to improve expected runtime bounds, and vice versa.

To improve $\mathcal{RB}_{\mathbb{E}}$, we generate a PLRF $\mathfrak{r}$ for a part of the program with the general transitions $\mathcal{GT}_{\text{ni}}$. To obtain a bound for the *full* program from $\mathfrak{r}$, for any $\ell \in \mathcal{L}$ let its *entry transitions* $\mathcal{ET}_{\mathcal{GT}_{\text{ni}}}(\ell)$ be those transitions from $\mathcal{GT} \setminus \mathcal{GT}_{\text{ni}}$ that can enter ℓ, and let $\mathcal{GT}_{\text{ni}}$'s *entry locations* $\mathcal{EL}_{\mathcal{GT}_{\text{ni}}}$ be those start locations of $\mathcal{GT}_{\text{ni}}$ whose entry transitions are not empty.

▶ **Example 5** (Entry Locations and Transitions). *For the PIP from Fig. 1 and* $\mathcal{GT}_{\text{ni}} = \{g_1\}$*, we have* $\mathcal{EL}_{\mathcal{GT}_{\text{ni}}} = \{\ell_1\}$ *and* $\mathcal{ET}_{\mathcal{GT}_{\text{ni}}}(\ell_1) = \{g_0\}$*. So the loop formed by* g_1 *is entered at* ℓ_1*, and* g_0 *is executed before. Similarly, for* $\mathcal{GT}'_{\text{ni}} = \{g_3\}$ *we have* $\mathcal{EL}_{\mathcal{GT}'_{\text{ni}}} = \{\ell_2\}$ *and* $\mathcal{ET}_{\mathcal{GT}'_{\text{ni}}}(\ell_2) = \{g_2\}$*.*

Recall that if $\mathfrak{r}$ is a PLRF for $\mathcal{GT}_> \subseteq \mathcal{GT}_{\text{ni}}$, then in a program that is restricted to $\mathcal{GT}_{\text{ni}}$, $\mathfrak{r}(\ell)$ is an upper bound on the expected number of executions of transitions from $\mathcal{GT}_>$ when starting in ℓ. Since $\mathfrak{r}(\ell)$ may contain negative coefficients, it is not weakly monotonically increasing in general. To transform polynomials into bounds from $\mathcal{B}$, let the over-approximation $\lceil \cdot \rceil$ replace all coefficients by their absolute value. So for example, $\lceil x - y \rceil = \lceil x + (-1) \cdot y \rceil = x + y$.

To turn $\lceil \mathfrak{r}(\ell) \rceil$ into a bound for the full program, one has to take into account how often the sub-program $\mathcal{GT}_{\text{ni}}$ is reached via an entry transition $h \in \mathcal{ET}_{\mathcal{GT}_{\text{ni}}}(\ell)$ for some $\ell \in \mathcal{EL}_{\mathcal{GT}_{\text{ni}}}$. This can be over-approximated by $\sum_{t=(\ldots,\ell) \in h} \mathcal{RB}(t)$, which is an upper bound on the number of times that transitions in h to the entry location ℓ of $\mathcal{GT}_{\text{ni}}$ are applied in a full program run.

The bound $\lceil \mathfrak{r}(\ell) \rceil$ is expressed in terms of the program variables at the entry location ℓ of $\mathcal{GT}_{\text{ni}}$. To obtain a bound in terms of the variables at the start of the program, one has to take into account which value a program variable x may be expected to have when the sub-program $\mathcal{GT}_{\text{ni}}$ is reached. For every entry transition $h \in \mathcal{ET}_{\mathcal{GT}_{\text{ni}}}(\ell)$, this value can be over-approximated by $\mathcal{SB}_{\mathbb{E}}(h, \ell, x)$. Thus, we have to instantiate each variable x in $\lceil \mathfrak{r}(\ell) \rceil$ by $\mathcal{SB}_{\mathbb{E}}(h, \ell, x)$. Let $\mathcal{SB}_{\mathbb{E}}(h, \ell, \cdot) : \mathcal{V} \to \mathcal{B}$ be the mapping with $\mathcal{SB}_{\mathbb{E}}(h, \ell, \cdot)(x) = \mathcal{SB}_{\mathbb{E}}(h, \ell, x)$. Hence, $\mathcal{SB}_{\mathbb{E}}(h, \ell, \cdot)(\lceil \mathfrak{r}(\ell) \rceil)$ over-approximates the expected number of applications of $\mathcal{GT}_>$ if $\mathcal{GT}_{\text{ni}}$ is entered in location ℓ, where this bound is expressed in terms of the input variables of the program. Here, weak monotonic increase of $\lceil \mathfrak{r}(\ell) \rceil$ ensures that instantiating its variables by an over-approximation of their size yields an over-approximation of the runtime.

▶ **Theorem 6** (Expected Runtime Bounds). *Let* $(\mathcal{RB}_{\mathbb{E}}, \mathcal{SB}_{\mathbb{E}})$ *be an expected bound pair,* $\mathcal{RB}$ *a (non-probabilistic) runtime bound, and* $\mathfrak{r}$ *a PLRF for* $\mathcal{GT}_> \subseteq \mathcal{GT}_{\text{ni}} \subseteq \mathcal{GT}$*. Then* $\mathcal{RB}'_{\mathbb{E}} \colon \mathcal{GT} \to \mathcal{B}$ *is an expected runtime bound where*

$$\mathcal{RB}'_{\mathbb{E}}(g) = \begin{cases} \sum\limits_{\substack{\ell \in \mathcal{EL}_{\mathcal{GT}_{\mathrm{ni}}} \\ h \in \mathcal{ET}_{\mathcal{GT}_{\mathrm{ni}}}(\ell)}} \Big(\sum\limits_{t=(\ldots,\ell)\in h} \mathcal{RB}(t) \Big) \cdot \left(\mathcal{SB}_{\mathbb{E}}(h,\ell,\cdot)\left(\lceil \mathfrak{r}(\ell) \rceil\right)\right), & \text{if } g \in \mathcal{GT}_{>} \\ \mathcal{RB}_{\mathbb{E}}(g), & \text{if } g \notin \mathcal{GT}_{>} \end{cases}$$

▶ **Example 7** (Expected Runtime Bounds). *For the PIP from Fig. 1, our approach starts with lifting the bound pair from Ex. 1 which results in* $\mathcal{RB}_{\mathbb{E}}(g_0) = \mathcal{RB}_{\mathbb{E}}(g_2) = 1$ *and* $\mathcal{RB}_{\mathbb{E}}(g_1) = \mathcal{RB}_{\mathbb{E}}(g_3) = \infty$. *Moreover,* $\mathcal{SB}_{\mathbb{E}}(g_0,\ell_1,x) = x$, $\mathcal{SB}_{\mathbb{E}}(g_1,\ell_1,x) = 2\cdot x$, $\mathcal{SB}_{\mathbb{E}}(g_2,\ell_2,x) = \mathcal{SB}_{\mathbb{E}}(g_3,\ell_2,x) = 3\cdot x$, $\mathcal{SB}_{\mathbb{E}}(g_0,\ell_1,y) = y$, *and* $\mathcal{SB}_{\mathbb{E}}(g,\_,y) = \infty$ *whenever* $g \neq g_0$.

To improve the bound $\mathcal{RB}_{\mathbb{E}}(g_1) = \infty$, *we use the PLRF* $\mathfrak{r}$ *for* $\mathcal{GT}_{>} = \mathcal{GT}_{\mathrm{ni}} = \{g_1\}$ *from Ex. 4. By Ex. 5, we have* $\mathcal{EL}_{\mathcal{GT}_{\mathrm{ni}}} = \{\ell_1\}$ *and* $\mathcal{ET}_{\mathcal{GT}_{\mathrm{ni}}}(\ell_1) = \{g_0\}$ *with* $g_0 = \{t_0\}$, *whose runtime bound is* $\mathcal{RB}(t_0) = 1$, *see Ex. 1. Using the expected size bound* $\mathcal{SB}_{\mathbb{E}}(g_0,\ell_1,x) = x$ *from Ex. 3, Thm. 6 yields* $\mathcal{RB}'_{\mathbb{E}}(g_1) = \mathcal{RB}(t_0)\cdot\mathcal{SB}_{\mathbb{E}}(g_0,\ell_1,\cdot)\left(\lceil \mathfrak{r}(\ell_1)\rceil\right) = 1\cdot 2\cdot x = 2\cdot x$. *To improve* $\mathcal{RB}_{\mathbb{E}}(g_3)$, *we use the PLRF* $\mathfrak{r}'$ *for* $\mathcal{GT}'_{>} = \mathcal{GT}'_{\mathrm{ni}} = \{g_3\}$ *from Ex. 4. As* $\mathcal{EL}_{\mathcal{GT}'_{\mathrm{ni}}} = \{\ell_2\}$ *and* $\mathcal{ET}_{\mathcal{GT}'_{\mathrm{ni}}}(\ell_2) = \{g_2\}$ *by Ex. 5, where* $g_2 = \{t_3\}$ *and* $\mathcal{RB}(t_3) = 1$ *(Ex. 1), with the bound* $\mathcal{SB}_{\mathbb{E}}(g_2,\ell_2,y) = 6\cdot x^2 + 2\cdot y$ *from Ex. 3, Thm. 6 yields* $\mathcal{RB}'_{\mathbb{E}}(g_3) = \mathcal{RB}(t_3)\cdot\mathcal{SB}_{\mathbb{E}}(g_2,\ell_2,\cdot)\left(\lceil \mathfrak{r}'(\ell_2)\rceil\right) = 1\cdot\mathcal{SB}_{\mathbb{E}}(g_2,\ell_2,y) = 6\cdot x^2 + 2\cdot y$. *So based on the expected size bounds of Ex. 3, we have shown how to compute the expected runtime bounds of Ex. 3 automatically. As mentioned, we refer to [5] for the computation of (improved) expected size bounds.*

Our approach relies on combining bounds that one has computed earlier in order to derive new bounds. If a new bound is computed by *linear combinations* of earlier bounds, then it is sound to use the "expected versions" of these earlier bounds, while this is not the case if two bounds are *multiplied*. Thus, it would be *unsound* to use the *expected* runtime bounds $\mathcal{RB}_{\mathbb{E}}(h)$ instead of the *non-probabilistic* bounds $\sum_{t=(\ldots,\ell)\in h}\mathcal{RB}(t)$ on the entry transitions in Thm. 6. If bounds $b_1,\ldots,b_n$ are *substituted* into another bound b, then it is sound to use "expected versions" of $b_1,\ldots,b_n$ if b is *concave*, see, e.g., [1]. Since bounds from $\mathcal{B}$ do not contain negative coefficients, we obtain that a finite bound $b \in \mathcal{B}$ is concave iff it is a linear polynomial. Thus, in Thm. 6 we may substitute *expected* size bounds $\mathcal{SB}_{\mathbb{E}}(h,\ell,x)$ into $\lceil \mathfrak{r}(\ell)\rceil$, since we restricted ourselves to *linear* ranking functions $\mathfrak{r}$.

4 Implementation and Conclusion

We presented a new modular approach to infer upper bounds on the expected runtimes of probabilistic integer programs. To this end, non-probabilistic and expected runtime and size bounds on parts of the program are computed in an alternating fashion and then combined to an overall expected runtime bound.

We implemented our analysis in a new version of our open-source tool KoAT [3], which can also be accessed via a web interface [4]. To assess the power of our approach, we compared KoAT with the main other related tools Absynth [6] and eco-imp [1] which are both based on a conceptionally different *backward-reasoning* approach. We ran the tools on a collection of 75 examples from Absynth's evaluation in [6] and additional benchmarks, including 10 larger PIPs based on benchmarks from the *TPDB* [7], where some transitions were enriched with probabilistic behavior. In the evaluation, KoAT succeeded on 91% of the examples, while Absynth and eco-imp only inferred finite bounds for 68% resp. 77% of the examples. For further details on the experiments and an overview on related work, we refer to [5].

References

1 M. Avanzini, G. Moser, and M. Schaper. A modular cost analysis for probabilistic programs. *Proc. ACM Program. Lang.*, 4(OOPSLA), 2020. doi:10.1145/3428240.

2 O. Bournez and F. Garnier. Proving positive almost-sure termination. In *Proc. RTA '05*, LNCS 3467, pages 323–337, 2005. doi:10.1007/978-3-540-32033-3_24.

3 M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Analyzing runtime and size complexity of integer programs. *ACM TOPLAS*, 38(4), 2016. doi:10.1145/2866575.

4 KoAT: https://aprove-developers.github.io/ExpectedUpperBounds/.

5 F. Meyer, M. Hark, and J. Giesl. Inferring expected runtimes of probabilistic integer programs using expected sizes. In *Proc. TACAS '21*, LNCS 12651, pages 250–269, 2021. Long version at https://arxiv.org/abs/2010.06367. doi:10.1007/978-3-030-72016-2_14.

6 V. C. Ngo, Q. Carbonneaux, and J. Hoffmann. Bounded expectations: Resource analysis for probabilistic programs. In *Proc. PLDI '18,* pp. 496–512, 2018. doi:10.1145/3192366.3192394.

7 TPDB: http://termination-portal.org/wiki/TPDB.

Parallel Complexity of Term Rewriting Systems*

Thaïs Baudon
ENS de Rennes & LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA), Lyon, France

Carsten Fuhs
Birkbeck, University of London, United Kingdom

Laure Gonnord
University of Lyon & LIP (UMR CNRS/ENS Lyon/UCB Lyon1/INRIA), Lyon, France

Abstract

In this workshop paper, we revisit the notion of parallel-innermost term rewriting. We provide a definition of parallel complexity and propose techniques to derive upper bounds on this complexity via the Dependency Tuple framework by Noschinski et al.

2012 ACM Subject Classification Theory of computation → Program verification, Rewrite systems; Software and its engineering → Automated static analysis, Formal software verification

Keywords and phrases Complexity analysis, Parallelism, Rewriting

1 Introduction

In this extended abstract, we consider the problem of evaluating the potentiality of parallelisation in pattern-matching based recursive functions like the one depicted in Figure 1.

```
fn size(&self) -> int {
  match self {
    &Tree::Node { v, ref left, ref right }
    =>  left.size() + right.size() + 1,
    &Tree::Empty => 0 , }    }
```

Figure 1 Tree size computation in Rust

In this particular example, the recursive calls to `left.size()` and `right.size()` can be done in parallel. Building on previous work on parallel-innermost rewriting [7, 4], and first ideas about parallel complexity [1], we propose a new notion of Parallel Dependency Tuples that capture such a behaviour, and a method to compute *parallel complexity bounds*.

2 Parallel-innermost Term Rewriting

The following definitions are mostly standard [3].

▶ **Definition 1** (Term rewrite system, innermost rewriting). *$\mathcal{T}(\Sigma, \mathcal{V})$ denotes the set of* terms *over a finite signature Σ and the set of variables $\mathcal{V}$. For a term t, the set $\mathcal{P}os(t)$ of its* positions *is defined inductively as a set of strings of positive integers: (a) if $t \in \mathcal{V}$, then $\mathcal{P}os(t) = \{\varepsilon\}$, and (b) if $t = f(t_1, \ldots, t_n)$, then $\mathcal{P}os(t) = \{\varepsilon\} \cup \bigcup_{1 \leq i \leq n} \{i\pi \mid \pi \in \mathcal{P}os(t_i)\}$. The position ε is called the* root position *of term t. The* (strict) prefix order $<$ *on positions is the strict partial order given by: $\pi < \tau$ iff there exists $\pi' \neq \varepsilon$ such that $\pi\pi' = \tau$. For*

* This work was partially funded by the French National Agency of Research in the CODAS Project (ANR-17-CE23-0004-01).

$\pi \in \mathcal{P}os(t)$, $t|_\pi$ is the subterm of t at position π, and we write $t[s]_\pi$ for the term that results from t by replacing the subterm $t|_\pi$ at position π by the term s.

For a term t, $\mathcal{V}(t)$ is the set of variables in t. If t has the form $f(t_1, \ldots, t_n)$, $\mathrm{root}(t) = f$ is the root *of t. A* term rewrite system (TRS) *$\mathcal{R}$ is a set of rules $\{\ell_1 \to r_1, \ldots, \ell_n \to r_n\}$ with $\ell_i, r_i \in \mathcal{T}(\Sigma, \mathcal{V})$, $\ell_i \notin \mathcal{V}$, and $\mathcal{V}(r_i) \subseteq \mathcal{V}(\ell_i)$ for all $1 \leq i \leq n$. The* rewrite relation *of $\mathcal{R}$ is $s \to_\mathcal{R} t$ iff there are a rule $\ell \to r \in \mathcal{R}$, a position $\pi \in \mathcal{P}os(s)$, and a substitution σ such that $s = s[\ell\sigma]_\pi$ and $t = s[r\sigma]_\pi$. Here, σ is called the* matcher *and the term $\ell\sigma$ is called the* redex *of the rewrite step. If $\ell\sigma$ has no proper subterm that is also a possible redex, $\ell\sigma$ is an* innermost redex*, and the rewrite step is an* innermost rewrite step *denoted by $s \xrightarrow{\mathsf{i}}_\mathcal{R} t$.*

$\Sigma_d^\mathcal{R} = \{f \mid f(\ell_1, \ldots, \ell_n) \to r \in \mathcal{R}\}$ and $\Sigma_c^\mathcal{R} = \Sigma \setminus \Sigma_d^\mathcal{R}$ are the defined *and* constructor *symbols of $\mathcal{R}$. We may omit the superscript and just write Σ_d and Σ_c if $\mathcal{R}$ is not of importance or clear from the context. Finally, let $\mathcal{P}os_d(t) = \{\pi \mid \pi \in \mathcal{P}os(t), \mathrm{root}(t|_\pi) \in \Sigma_d\}$.*

The notion of parallel-innermost rewriting dates back at least to [7]. Informally, in a parallel-innermost rewrite step, all innermost redexes are rewritten simultaneously. This corresponds to executing all function calls in parallel on a machine with unbounded parallelism.

▶ **Definition 2** (Parallel-innermost rewriting [4]). *A term s* rewrites innermost in parallel *to t with a TRS $\mathcal{R}$, written $s \Vdash^{\mathsf{i}}_\mathcal{R} t$, iff $s \xrightarrow{\mathsf{i}}{}^+_\mathcal{R} t$, and either (a) $s \xrightarrow{\mathsf{i}}_\mathcal{R} t$ with s an innermost redex, or (b) $s = f(s_1, \ldots, s_n)$, $t = f(t_1, \ldots, t_n)$, and for all $1 \leq k \leq n$ either $s_k \Vdash^{\mathsf{i}}_\mathcal{R} t_k$ or $s_k = t_k$ is a normal form.*

▶ **Example 3** (size). Consider the TRS $\mathcal{R}$ with the following rules modelling the code of Figure 1.

$$\begin{array}{lcl} \mathsf{plus}(\mathsf{Zero}, y) & \to & y \\ \mathsf{plus}(\mathsf{S}(x), y) & \to & \mathsf{S}(\mathsf{plus}(x, y)) \end{array} \qquad \begin{array}{lcl} \mathsf{size}(\mathsf{Nil}) & \to & \mathsf{Zero} \\ \mathsf{size}(\mathsf{Tree}(v, l, r)) & \to & \mathsf{S}(\mathsf{plus}(\mathsf{size}(l), \mathsf{size}(r))) \end{array}$$

Here $\Sigma_d^\mathcal{R} = \{\mathsf{plus}, \mathsf{size}\}$ and $\Sigma_c^\mathcal{R} = \{\mathsf{Zero}, \mathsf{S}, \mathsf{Nil}, \mathsf{Tree}\}$. We have the following parallel innermost rewrite sequence, where innermost redexes are underlined:

$$\begin{array}{ll} & \underline{\mathsf{size}(\mathsf{Tree}(\mathsf{Zero}, \mathsf{Nil}, \mathsf{Tree}(\mathsf{Zero}, \mathsf{Nil}, \mathsf{Nil})))} \\ \Vdash^{\mathsf{i}}_\mathcal{R} & \mathsf{S}(\mathsf{plus}(\underline{\mathsf{size}(\mathsf{Nil})}, \underline{\mathsf{size}(\mathsf{Tree}(\mathsf{Zero}, \mathsf{Nil}, \mathsf{Nil}))})) \\ \Vdash^{\mathsf{i}}_\mathcal{R} & \mathsf{S}(\mathsf{plus}(\mathsf{Zero}, \mathsf{S}(\mathsf{plus}(\underline{\mathsf{size}(\mathsf{Nil})}, \underline{\mathsf{size}(\mathsf{Nil})})))) \\ \Vdash^{\mathsf{i}}_\mathcal{R} & \mathsf{S}(\mathsf{plus}(\mathsf{Zero}, \mathsf{S}(\underline{\mathsf{plus}(\mathsf{Zero}, \mathsf{Zero})}))) \\ \Vdash^{\mathsf{i}}_\mathcal{R} & \mathsf{S}(\underline{\mathsf{plus}(\mathsf{Zero}, \mathsf{S}(\mathsf{Zero}))}) \\ \Vdash^{\mathsf{i}}_\mathcal{R} & \mathsf{S}(\mathsf{S}(\mathsf{Zero})) \end{array}$$

Note that in the second and in the third step, two innermost steps each are happening in parallel. A corresponding regular innermost rewrite sequence without parallel evaluation of redexes would have needed two more steps.

3 Finding Upper Bounds for Parallel Complexity

3.1 Notion of Parallel Complexity

We extend the notion of innermost runtime complexity to parallel-innermost rewriting.

▶ **Definition 4** ((Parallel) Innermost Runtime Complexity). *The* size *$|t|$ of a term t is $|x| = 1$ if $x \in \mathcal{V}$ and $|f(t_1, \ldots, t_n)| = 1 + \sum_{i=1}^n |t_i|$, otherwise. The* derivation height *of a term t w.r.t. a relation $\to$ is the length of the longest sequence of $\to$-steps from t: $\mathrm{dh}(t, \to) = \sup\{e \mid \exists\, t' \in \mathcal{T}(\Sigma, \mathcal{V}).\ t \to^e t'\}$ where $\to^e$ is the e^{th} iterate of $\to$. If t starts an infinite $\to$-sequence, we write $\mathrm{dh}(t, \to) = \omega$.*

A term $f(t_1,\dots,t_k)$ *is* basic (for a TRS $\mathcal{R}$) *iff* $f \in \Sigma_d^{\mathcal{R}}$ *and* $t_1,\dots,t_k \in \mathcal{T}(\Sigma_c^{\mathcal{R}},\mathcal{V})$. $\mathcal{T}_{\text{basic}}^{\mathcal{R}}$ *is the set of basic terms for a TRS* $\mathcal{R}$. *For* $n \in \mathbb{N}$, *we define the* innermost runtime complexity *function* $\text{irc}_{\mathcal{R}}(n) = \sup\{\text{dh}(t, \xrightarrow{i}_{\mathcal{R}}) \mid t \in \mathcal{T}_{\text{basic}}, |t| \le n\}$ *and we introduce the* parallel innermost runtime complexity *function* $\text{irc}_{\mathcal{R}}^{\parallel}(n) = \sup\{\text{dh}(t, \overset{i}{\parallel\!\!\rightarrow}_{\mathcal{R}}) \mid t \in \mathcal{T}_{\text{basic}}, |t| \le n\}$.

In the following, given a TRS $\mathcal{R}$, our goal shall be to infer (asymptotic) upper bounds for $\text{irc}_{\mathcal{R}}^{\parallel}$ fully automatically. As usual for runtime complexity, we are considering only basic terms as start terms, corresponding to a defined function called on data objects as arguments. An upper bound for (sequential) $\text{irc}_{\mathcal{R}}$ is also an upper bound for $\text{irc}_{\mathcal{R}}^{\parallel}$. We will introduce techniques to find upper bounds for $\text{irc}_{\mathcal{R}}^{\parallel}$ that are strictly tighter than these trivial bounds.

3.2 Complexity: the sequential case

We build on the Dependency Tuple framework [6], originally introduced to determine upper bounds for (sequential) innermost runtime complexity. A central idea is to group all function calls by a rewrite rule *together* rather than to regard them separately (as for termination [2]).

▶ **Definition 5** (Sharp Terms $\mathcal{T}^\sharp$). *For every* $f \in \Sigma_d$, *we introduce a fresh symbol* $f^\sharp$ *of the same arity. For a term* $t = f(t_1,\dots,t_n)$ *with* $f \in \Sigma_d$, *we define* $t^\sharp = f^\sharp(t_1,\dots,t_n)$ *and let* $\mathcal{T}^\sharp = \{t^\sharp \mid t \in \mathcal{T}(\Sigma,\mathcal{V}), \text{root}(t) \in \Sigma_d\}$.

To compute an upper bound for sequential complexity, we "count" how often each rewrite rule is used. The idea is that the cost of the function call to the lhs of a rule is 1 + the sum of the costs of all the function calls in the rhs, counted separately. To group k function calls together, we use "compound symbols" Com_k, which intuitively represent the sum of the runtimes of their arguments. Then, we can use polynomial interpretations $\mathcal{P}ol$ with $\mathcal{P}ol(\mathsf{Com}_k(x_1,\dots,x_k)) = x_1 + \dots + x_k$ for all k to compute a complexity bound [6, Thm. 27].

▶ **Definition 6** (Dependency Tuple, DT [6]). *A* dependency tuple (DT) *is a rule of the form* $s^\sharp \to \mathsf{Com}_n(t_1^\sharp,\dots,t_n^\sharp)$ *where* $s^\sharp, t_1^\sharp,\dots,t_n^\sharp \in \mathcal{T}^\sharp$. *Let* $\ell \to r$ *be a rule with* $\mathcal{P}os_d(r) = \{\pi_1,\dots,\pi_n\}$ *and* $\pi_1 \lhd \dots \lhd \pi_n$ *for a total order* $\lhd$ *on positions. Then* $DT(\ell \to r) = \ell^\sharp \to \mathsf{Com}_n(r|_{\pi_1}^\sharp,\dots,r|_{\pi_n}^\sharp)$. *For a TRS* $\mathcal{R}$, *let* $DT(\mathcal{R}) = \{DT(\ell \to r) \mid \ell \to r \in \mathcal{R}\}$.

▶ **Example 7.** For our running example, we get the following DTs:

$$\begin{aligned}
\mathsf{plus}^\sharp(\mathsf{Zero}, y) &\to \mathsf{Com}_0\\
\mathsf{plus}^\sharp(\mathsf{S}(x), y) &\to \mathsf{Com}_1(\mathsf{plus}^\sharp(x,y))\\
\mathsf{size}^\sharp(\mathsf{Nil}) &\to \mathsf{Com}_0\\
\mathsf{size}^\sharp(\mathsf{Tree}(v,l,r)) &\to \mathsf{Com}_3(\mathsf{size}^\sharp(l), \mathsf{size}^\sharp(r), \mathsf{plus}^\sharp(\mathsf{size}(l), \mathsf{size}(r)))
\end{aligned}$$

The following polynomial interpretation, which orients all DTs with $\succ$ and all rules from $\mathcal{R}$ with $\succsim$, proves $\text{irc}_{\mathcal{R}}(n) \in \mathcal{O}(n^2)$: $\mathcal{P}ol(\mathsf{plus}^\sharp(x_1,x_2)) = \mathcal{P}ol(\mathsf{size}(x_1)) = x_1, \mathcal{P}ol(\mathsf{size}^\sharp(x_1)) = 2x_1 + x_1^2, \mathcal{P}ol(\mathsf{plus}(x_1,x_2)) = x_1 + x_2, \mathcal{P}ol(\mathsf{Tree}(x_1,x_2,x_3)) = 1 + x_2 + x_3, \mathcal{P}ol(\mathsf{S}(x_1)) = 1 + x_1, \mathcal{P}ol(\mathsf{Zero}) = \mathcal{P}ol(\mathsf{Nil}) = 1$. Since for all constructor symbols f, $\mathcal{P}ol(f(x_1,\dots,x_n)) \le x_1 + \dots + x_n + c$ for some $c \in \mathbb{N}$ and since the maximal degree of the polynomial interpretation is 2, the upper bound of $\mathcal{O}(n^2)$ follows by [6, Thm. 27].

3.3 Computing Upper Bounds for Parallel Rewriting

To find upper bounds for runtime complexity of parallel-innermost rewriting, we can *reuse* the notion of DTs from Def. 6 for sequential innermost rewriting along with existing techniques [6] and implementations. We illustrate this in the following example.

▶ **Example 8.** In the recursive size-rule, the two calls to $\mathsf{size}(l)$ and $\mathsf{size}(r)$ happen *in parallel* (this will be captured by the notion of *structural independency*). Thus, the cost for these two calls is not the *sum*, but the *maximum* of the calls. Regardless of which of these two calls has the higher cost, we still need to add the cost for the call to plus on the results of the two calls, which starts evaluating only after both calls to size have finished. With σ as the used matcher for the rule and with $t\!\downarrow$ as the (here unique) normal form resulting from repeatedly rewriting a term t with $\overset{\mathsf{i}}{\Longrightarrow}_{\mathcal{R}}$ (the "result" of evaluating t), we have:

$$\begin{aligned} & \mathrm{dh}(\mathsf{size}(\mathsf{Tree}(v,l,r))\sigma, \overset{\mathsf{i}}{\Longrightarrow}_{\mathcal{R}}) \\ = \;& 1 + \max(\mathrm{dh}(\mathsf{size}(l)\sigma, \overset{\mathsf{i}}{\Longrightarrow}_{\mathcal{R}}), \mathrm{dh}(\mathsf{size}(r)\sigma, \overset{\mathsf{i}}{\Longrightarrow}_{\mathcal{R}})) + \mathrm{dh}(\mathsf{plus}(\mathsf{size}(l)\sigma\!\downarrow, \mathsf{size}(r)\sigma\!\downarrow), \overset{\mathsf{i}}{\Longrightarrow}_{\mathcal{R}}) \end{aligned}$$

We could now introduce a new symbol ComPar_k that explicitly expresses that its arguments are evaluated in parallel. This symbol would then be interpreted as the maximum of its arguments in an extension of [6, Thm. 27]:

$$\mathsf{size}^\sharp(\mathsf{Tree}(v,l,r)) \to \mathsf{Com}_2(\mathsf{ComPar}_2(\mathsf{size}^\sharp(l), \mathsf{size}^\sharp(r)), \mathsf{plus}^\sharp(\mathsf{size}(l), \mathsf{size}(r)))$$

Although automation of the search for polynomial implementations extended by the maximum function is readily available [5], we would still have to extend the notion of Dependency Tuples and also adapt all existing techniques in the Dependency Tuple Framework to work with ComPar_k.

This is why we have chosen the following alternative approach. Equivalently to the above, we can "factor in" the cost of calling plus into the maximum function:

$$\begin{aligned} & \mathrm{dh}(\mathsf{size}(\mathsf{Tree}(v,l,r))\sigma, \overset{\mathsf{i}}{\Longrightarrow}_{\mathcal{R}}) \\ = \;& \max(1 + \mathrm{dh}(\mathsf{size}(l)\sigma, \overset{\mathsf{i}}{\Longrightarrow}_{\mathcal{R}}) + \mathrm{dh}(\mathsf{plus}(\mathsf{size}(l)\sigma\!\downarrow, \mathsf{size}(r)\sigma\!\downarrow), \overset{\mathsf{i}}{\Longrightarrow}_{\mathcal{R}}), \\ & \quad 1 + \mathrm{dh}(\mathsf{size}(r)\sigma, \overset{\mathsf{i}}{\Longrightarrow}_{\mathcal{R}}) + \mathrm{dh}(\mathsf{plus}(\mathsf{size}(l)\sigma\!\downarrow, \mathsf{size}(r)\sigma\!\downarrow), \overset{\mathsf{i}}{\Longrightarrow}_{\mathcal{R}})) \end{aligned}$$

Intuitively, this would correspond to evaluating $\mathsf{plus}(\mathsf{size}(l), \mathsf{size}(r))$ twice, in two parallel threads of execution, which costs the same amount of time as evaluating $\mathsf{plus}(\mathsf{size}(l), \mathsf{size}(r))$ once. We can represent this maximum of the execution times of two threads by introducing *two* DTs for our recursive size-rule:

$$\begin{aligned} \mathsf{size}^\sharp(\mathsf{Tree}(v,l,r)) &\to \mathsf{Com}_2(\mathsf{size}^\sharp(l), \mathsf{plus}^\sharp(\mathsf{size}(l), \mathsf{size}(r))) \\ \mathsf{size}^\sharp(\mathsf{Tree}(v,l,r)) &\to \mathsf{Com}_2(\mathsf{size}^\sharp(r), \mathsf{plus}^\sharp(\mathsf{size}(l), \mathsf{size}(r))) \end{aligned}$$

To express the cost of a concrete rewrite sequence, we would non-deterministically choose the DT that corresponds to the "slower thread".

In other words, the cost of the function call to the lhs of a rule is 1 + the sum of the costs of all the function calls in the rhs *that are in structural dependency with each other*. The actual cost of the function call to the lhs in a concrete rewrite sequence is the *maximum* of all the possible costs caused by such *chains* of structural dependency (based on the prefix order $>$ on positions of defined function symbols in the rhs). Thus, *structurally independent* function calls are considered in separate DTs, whose non-determinism models the parallelism of these function calls.

The notion of *structural dependency* of function calls is captured by Def. 9. Basically, it comes from the fact that a term cannot be evaluated before all its subterms have been reduced to normal forms (innermost rewriting/*call by value*). This induces a "happens-before" relation for the computation.

▶ **Definition 9** (Structural dependency)**.** *Let t be a term and τ_1, τ_2 be the positions of two defined symbols in t. Let $t_1 = t|_{\tau_1}$ and $t_2 = t|_{\tau_2}$. Then t_1* structurally depends *on t_2 iff $\tau_1 < \tau_2$ in the prefix order $<$ (i.e., t_2 is a subterm of t_1).*

▶ **Example 10.** Let $t = \mathsf{S}(\mathsf{plus}(\mathsf{size}(\mathsf{Nil}), \mathsf{plus}(\mathsf{size}(x), \mathsf{Zero})))$. For our running example, we find the following structural dependencies in t:

- The term $t|_1 = \mathsf{plus}(\mathsf{size}(\mathsf{Nil}), \mathsf{plus}(\mathsf{size}(x), \mathsf{Zero}))$ structurally depends on $t|_{11} = \mathsf{size}(\mathsf{Nil})$, on $t|_{12} = \mathsf{plus}(\mathsf{size}(x), \mathsf{Zero})$, and on $t|_{121} = \mathsf{size}(x)$.
- The term $t|_{12} = \mathsf{plus}(\mathsf{size}(x), \mathsf{Zero})$ structurally depends on $t|_{121} = \mathsf{size}(x)$.

It is worth noting that when s structurally depends on t, neither s nor t need to be a redex – what matters for our purposes is that t could be *instantiated* to a (potential) redex and that an instance of s could become a redex after its subterms, including the instance of t, have been fully evaluated. It is also worth noting that the structural dependency relation is transitive.

We thus revisit the notion of DTs, which now embed structural dependencies.

▶ **Definition 11** (Parallel Dependency Tuples $DT^{\|}$, PDTs). *For a rewrite rule $\ell \to r$, we define the set of its* Parallel Dependency Tuples (PDTs) $DT^{\|}(\ell \to r)$*: if $\mathcal{P}os_d(r) = \emptyset$, then $DT^{\|}(\ell \to r) = \{\ell^\sharp \to \mathsf{Com}_0\}$; otherwise, $DT^{\|}(\ell \to r) = \{\ell^\sharp \to \mathsf{Com}_k(r|^\sharp_{\pi_1}, \ldots, r|^\sharp_{\pi_k}) \mid k > 0, \pi_1 > \cdots > \pi_k$ is a maximal structural dependency chain in $\mathcal{P}os_d(r)\}$, where $\pi_1 > \cdots > \pi_k$ is a maximal structural dependency chain in $\mathcal{P}os_d(r)$ iff $\forall \pi \in \mathcal{P}os_d(r), \pi \not> \pi_1 \wedge \pi_k \not> \pi$. For a TRS $\mathcal{R}$, let $DT^{\|}(\mathcal{R}) = \bigcup_{\ell \to r \in \mathcal{R}} DT^{\|}(\ell \to r)$.*

▶ **Example 12.** For our recursive size-rule $lhs \to rhs$, we have $\mathcal{P}os_d(rhs) = \{1, 11, 12\}$. The two maximal $>$-chains are $11 > 1$ and $12 > 1$. With $rhs|_1 = \mathsf{plus}(\mathsf{size}(l), \mathsf{size}(r))$, $rhs|_{11} = \mathsf{size}(l)$, and $rhs|_{12} = \mathsf{size}(r)$, we get the PDTs from Ex. 8.

To connect PDTs with our parallel-innermost rewrite relation $\overset{\|\mathsf{i}}{\to}_{\mathcal{R}}$, we need the notion of *chain tree*, which is an extension of dependency chains [2], and its complexity.

▶ **Definition 13** (Chain Tree, *Cplx* [6]). *Let $\mathcal{D}$ be a set of DTs and $\mathcal{R}$ be a TRS. Let T be a (possibly infinite) tree whose nodes are labelled with a DT from $\mathcal{D}$ and a substitution. Let the root node be labelled with $(s^\sharp \to \mathsf{Com}_n(\ldots) \mid \sigma)$. Then T is a $(\mathcal{D}, \mathcal{R})$-*chain tree for $s^\sharp\sigma$ *iff the following conditions hold for any node of T, where $(u^\sharp \to \mathsf{Com}_m(v^\sharp_1, \ldots, v^\sharp_m) \mid \mu)$ is the label of the node:*

- *$u^\sharp\mu$ is in normal form w.r.t. $\mathcal{R}$;*
- *if this node has the children $(p^\sharp_1 \to \mathsf{Com}_{m_1}(\ldots) \mid \delta_1), \ldots, (p^\sharp_k \to \mathsf{Com}_{m_k}(\ldots) \mid \delta_k)$, then there are pairwise different $i_1, \ldots, i_k \in \{1, \ldots, m\}$ with $v^\sharp_{i_j}\mu \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} p^\sharp_j\delta_j$ for all $j \in \{1, \ldots, k\}$.*

Let $\mathcal{S} \subseteq \mathcal{D}$ and $s^\sharp \in \mathcal{T}^\sharp$. For a chain tree T, $|T|_{\mathcal{S}} \in \mathbb{N} \cup \{\omega\}$ is the number of nodes in T labelled with a DT from $\mathcal{S}$. We define $Cplx_{\langle \mathcal{D}, \mathcal{S}, \mathcal{R}\rangle}(s^\sharp) = \sup\{|T|_{\mathcal{S}} \mid T$ is a $(\mathcal{D}, \mathcal{R})$-chain tree for $s^\sharp\}$. For terms $s^\sharp$ without a $(\mathcal{D}, \mathcal{R})$-chain tree, we define $Cplx_{\langle \mathcal{D}, \mathcal{S}, \mathcal{R}\rangle}(s^\sharp) = 0$.

We can now make our main correctness claim:

▶ **Proposition 14** (*Cplx* bounds Derivation Height for $\overset{\|\mathsf{i}}{\to}_{\mathcal{R}}$). *Let $\mathcal{R}$ be a TRS, let $t = f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$ such that all t_i are in normal form (in particular, this includes all $t \in \mathcal{T}_{\text{basic}}$). Then we have $\mathrm{dh}(t, \overset{\|\mathsf{i}}{\to}_{\mathcal{R}}) \leq Cplx_{\langle DT^{\|}(\mathcal{R}), DT^{\|}(\mathcal{R}), \mathcal{R}\rangle}(t^\sharp)$.*

Thus, via [6, Thm. 27], in particular we can use polynomial interpretations in the DT framework for our PDTs to get upper bounds for $\mathrm{irc}^{\|}_{\mathcal{R}}$.

▶ **Example 15** (Ex. 8 continued). For our TRS $\mathcal{R}$ computing the size function on trees, we get the set $DT^{\|}(\mathcal{R})$ with the following PDTs:

$$\begin{array}{l|l} \mathsf{plus}^\sharp(\mathsf{Zero}, y) \to \mathsf{Com}_0 & \mathsf{size}^\sharp(\mathsf{Nil}) \to \mathsf{Com}_0 \\ \mathsf{plus}^\sharp(\mathsf{S}(x), y) \to \mathsf{Com}_1(\mathsf{plus}^\sharp(x, y)) & \mathsf{size}^\sharp(\mathsf{Tree}(v, l, r)) \to \mathsf{Com}_2(\mathsf{size}^\sharp(l), \mathsf{plus}^\sharp(\mathsf{size}(l), \mathsf{size}(r))) \\ & \mathsf{size}^\sharp(\mathsf{Tree}(v, l, r)) \to \mathsf{Com}_2(\mathsf{size}^\sharp(r), \mathsf{plus}^\sharp(\mathsf{size}(l), \mathsf{size}(r))) \end{array}$$

The interpretation $\mathcal{P}ol$ from Ex. 7 implies $\mathrm{irc}^{\|}_{\mathcal{R}}(n) \in \mathcal{O}(n^2)$. This bound is tight: consider $\mathsf{size}(t)$ for a comb-shaped tree t where the first argument of Tree is always Zero and the third is always Nil. The function plus, which needs time linear in its first argument, is called linearly often on data linear in the size of the start term. Due to the structural dependencies, these calls do not happen in parallel (so call $k+1$ to plus must wait for call k).

▶ **Example 16.** Note that $\mathrm{irc}^{\|}(n)$ can be asymptotically lower than $\mathrm{irc}(n)$, for instance for the TRS $\mathcal{R}$ with the following rules:

$$\begin{array}{rcl|rcl} \mathsf{doubles}(\mathsf{Zero}) & \to & \mathsf{Nil} & \mathsf{d}(\mathsf{Zero}) & \to & \mathsf{Zero} \\ \mathsf{doubles}(\mathsf{S}(x)) & \to & \mathsf{Cons}(\mathsf{d}(\mathsf{S}(x)), \mathsf{doubles}(x)) & \mathsf{d}(\mathsf{S}(x)) & \to & \mathsf{S}(\mathsf{S}(\mathsf{d}(x))) \end{array}$$

The upper bound $\mathrm{irc}_{\mathcal{R}}(n) \in \mathcal{O}(n^2)$ is tight: from a term $\mathsf{doubles}(\mathsf{S}(\mathsf{S}(\ldots\mathsf{S}(\mathsf{Zero})\ldots)))$, we get linearly many calls of the linear-time function d on arguments of size linear in the start term. However, the Parallel Dependency Tuples in this example are:

$$\begin{array}{rcl|rcl} \mathsf{doubles}^\sharp(\mathsf{Zero}) & \to & \mathsf{Com}_0 & \mathsf{d}^\sharp(\mathsf{Zero}) & \to & \mathsf{Com}_0 \\ \mathsf{doubles}^\sharp(\mathsf{S}(x)) & \to & \mathsf{Com}_1(\mathsf{d}^\sharp(\mathsf{S}(x))) & \mathsf{d}^\sharp(\mathsf{S}(x)) & \to & \mathsf{Com}_1(\mathsf{d}^\sharp(x)) \\ \mathsf{doubles}^\sharp(\mathsf{S}(x)) & \to & \mathsf{Com}_1(\mathsf{doubles}^\sharp(x)) & & & \end{array}$$

Then the following polynomial interpretation, which orients all DTs with $\succ$ and all rules from $\mathcal{R}$ with $\succsim$, proves $\mathrm{irc}^{\|}_{\mathcal{R}} \in \mathcal{O}(n)$: $\mathcal{P}ol(\mathsf{doubles}^\sharp(x_1)) = \mathcal{P}ol(\mathsf{d}(x_1)) = 2x_1, \mathcal{P}ol(\mathsf{d}^\sharp(x_1)) = x_1, \mathcal{P}ol(\mathsf{doubles}(x_1)) = \mathcal{P}ol(\mathsf{Cons}(x_1, x_2)) = \mathcal{P}ol(\mathsf{Zero}) = \mathcal{P}ol(\mathsf{Nil}) = 1, \mathcal{P}ol(\mathsf{S}(x_1)) = 1 + x_1$.

4 Conclusion

We have come up with a notion of parallel runtime complexity and a concrete algorithm to compute upper bounds on this complexity on TRSs. Future work includes practical design of *parallel rewriting engines* that infer their rewriting schedules from parallel dependency tuples (while taking into account the limitations of the underlying hardware platform). Another goal is the formalisation of complexity w.r.t. term *height* (considering terms as trees), which seems to be more practical for our parallelisation needs.

References

1 Christophe Alias, Carsten Fuhs, and Laure Gonnord. Estimation of Parallel Complexity with Rewriting Techniques. In *Proc. WST '16*, pages 2:1–2:5, 2016.

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

3 Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge Univ. Press, 1998.

4 Mirtha-Lina Fernández, Guillem Godoy, and Albert Rubio. Orderings for innermost termination. In *Proc. RTA '05*, pages 17–31, 2005.

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

6 Lars Noschinski, Fabian Emmes, and Jürgen Giesl. Analyzing innermost runtime complexity of term rewriting by dependency pairs. *J. Autom. Reason.*, 51(1):27–56, 2013.

7 Jean Vuillemin. Correct and optimal implementations of recursion in a simple programming language. *J. Comput. Syst. Sci.*, 9(3):332–354, 1974.

Between Derivational and Runtime Complexity

Carsten Fuhs ✉
Birkbeck, University of London, United Kingdom

Abstract

Derivational complexity of term rewriting considers the length of the longest rewrite sequence for arbitrary start terms, whereas runtime complexity restricts start terms to basic terms. Recently, there has been notable progress in automatic inference of upper and lower bounds for runtime complexity. I propose a novel transformation that lets an off-the-shelf tool for inference of upper or lower bounds for runtime complexity determine upper or lower bounds for derivational complexity as well. The approach is applicable to derivational complexity problems for innermost rewriting and for full rewriting. I have implemented the transformation in the tool AProVE and conducted an extensive experimental evaluation. My results indicate that bounds for derivational complexity can now be inferred for rewrite systems that have been out of reach for automated analysis thus far.

2012 ACM Subject Classification Theory of computation → Program verification, Rewrite systems; Software and its engineering → Automated static analysis, Formal software verification

Keywords and phrases term rewriting, derivational complexity, runtime complexity, static analysis

1 Introduction and Preliminaries

Term rewrite systems (TRSs) are a classic computational model both for equational reasoning and for evaluation of programs with user-defined data structures and recursion. A widely studied question for TRSs is that of their *complexity*, i.e., the length of their longest derivation (i.e., rewrite sequence) as a function of the size of the start term of the derivation. From a program analysis perspective, this corresponds to the worst-case time complexity of the TRS.

Commonly two "flavors" of complexity are used, which differ by the set of start terms. (1) The *derivational complexity* [9] of a TRS considers arbitrary terms as start terms, including terms with several (possibly nested) function calls. It is inspired by the notion of termination, which considers whether all rewrite sequences from arbitrary start terms terminate. Derivational complexity is a suitable measure for the number of rewrite steps needed for deciding the word problem in first-order equational reasoning via a terminating and confluent TRS to rewrite both sides of the conjectured equality to normal form.

(2) The *runtime complexity* [8] of a TRS considers only *basic terms* as start terms: intuitively, these are terms where a single function call is performed on constructor terms (i.e., data objects) as arguments. The motivation for this restriction comes from program analysis with an interest in the running time of a function on data objects.

As far as I am aware, the two strands of research on derivational and on runtime complexity have essentially stayed separate thus far. This paper proposes a transformation between TRSs such that the runtime complexity of the transformed TRS is the same as the derivational complexity of the original TRS, both for innermost rewriting and for full rewriting. An extended conference version of this paper with formal definitions, proofs of the theorems, and a discussion of related work in complexity analysis and transformation-based techniques was published in 2019 [6].

Preliminaries. Basic knowledge of term rewriting is assumed. We recapitulate (relative) term rewriting as well as the notions of derivational complexity and runtime complexity.

▶ **Definition 1** (Signature, term, term rewriting, defined symbol, constructor symbol, basic term)**.** *$\mathcal{T}(\Sigma, \mathcal{V})$ is the set of* terms *over signature Σ and variables $\mathcal{V}$. For a term t, $\mathcal{V}(t)$ is the set of variables occurring in t. A* TRS *$\mathcal{R}$ is a set of rules $\{\ell_1 \to r_1, \ldots, \ell_n \to r_n\}$ with*

$\ell_i, r_i \in \mathcal{T}(\Sigma, \mathcal{V})$, $\ell_i \notin \mathcal{V}$, *and* $\mathcal{V}(r_i) \subseteq \mathcal{V}(\ell_i)$ *for all* $1 \le i \le n$. *Its* rewrite relation *is given by* $s \to_{\mathcal{R}} t$ *iff there is a rule* $\ell \to r \in \mathcal{R}$, *a position* $\pi \in \mathcal{P}os(s)$, *and a substitution* σ *with* $s = s[\ell\sigma]_\pi$ *and* $t = s[r\sigma]_\pi$. *Here* $\ell\sigma$ *is the* redex *of the rewrite step.*

For two TRSs $\mathcal{R}$ *and* $\mathcal{S}$, $\mathcal{R}/\mathcal{S}$ *is a* relative TRS, *and its* rewrite relation $\to_{\mathcal{R}/\mathcal{S}}$ *is* $\to_{\mathcal{S}}^* \circ \to_{\mathcal{R}} \circ \to_{\mathcal{S}}^*$. *We define the* innermost *rewrite relation by* $s \xrightarrow{\text{i}}_{\mathcal{R}/\mathcal{S}} t$ *iff* $s \to_{\mathcal{S}}^* s' \to_{\mathcal{R}} s'' \to_{\mathcal{S}}^* t$ *for some terms* s', s'' *such that the proper subterms of the redexes of each step with* $\to_{\mathcal{S}}$ *or* $\to_{\mathcal{R}}$ *are in normal form w.r.t.* $\mathcal{R} \cup \mathcal{S}$. *We may write* $\to_{\mathcal{R}}$ *for* $\to_{\mathcal{R}/\emptyset}$ *and* $\xrightarrow{\text{i}}_{\mathcal{R}}$ *for* $\xrightarrow{\text{i}}_{\mathcal{R}/\emptyset}$.

$\Sigma_d^{\mathcal{R}\cup\mathcal{S}} = \{f \mid f(\ell_1, \ldots, \ell_n) \to r \in \mathcal{R} \cup \mathcal{S}\}$ *and* $\Sigma_c^{\mathcal{R}\cup\mathcal{S}} = \{f \mid f \in \Sigma$ *occurs in some rule* $\ell \to r \in \mathcal{R} \cup \mathcal{S}\} \setminus \Sigma_d^{\mathcal{R}\cup\mathcal{S}}$ *are the* defined *and* constructor *symbols of* $\mathcal{R}/\mathcal{S}$. *We write* $\Sigma^{\mathcal{R}\cup\mathcal{S}} = \Sigma_d^{\mathcal{R}\cup\mathcal{S}} \uplus \Sigma_c^{\mathcal{R}\cup\mathcal{S}}$ *and* $\mathcal{T}^{\mathcal{R}/\mathcal{S}} = \mathcal{T}(\Sigma^{\mathcal{R}\cup\mathcal{S}}, \mathcal{V})$. *A term* $f(t_1, \ldots, t_k)$ *is* basic (for a given relative TRS $\mathcal{R}/\mathcal{S}$) *iff* $f \in \Sigma_d^{\mathcal{R}\cup\mathcal{S}}$ *and* $t_1, \ldots, t_k \in \mathcal{T}(\Sigma_c^{\mathcal{R}\cup\mathcal{S}}, \mathcal{V})$. $\mathcal{T}_{\text{basic}}^{\mathcal{R}/\mathcal{S}}$ *denotes the set of basic terms for* $\mathcal{R}/\mathcal{S}$.

▶ **Definition 2** (Size, derivation height, derivational complexity dc, runtime complexity rc [8, 9, 14]). *The* size $|t|$ *of a term* t *is* $|x| = 1$ *if* $x \in \mathcal{V}$ *and* $|f(t_1, \ldots, t_k)| = 1 + \sum_{i=1}^k |t_i|$, *otherwise. The* derivation height *of a term* t *w.r.t. a relation* $\to$ *is the length of the longest sequence of* $\to$*-steps starting with* t, *i.e.,* $\text{dh}(t, \to) = \sup\{e \mid \exists\, t' \in \mathcal{T}(\Sigma, \mathcal{V}).\ t \to^e t'\}$ *where* $\to^e$ *denotes the* e^{th} *iterate of* $\to$. *If* t *starts an infinite* $\to$*-sequence, we write* $\text{dh}(t, \to) = \omega$.

We first introduce a generic complexity function compl *parameterized by a natural number* n, *a relation* $\to$, *and a set of start terms* $\mathcal{T}_0$*:* $\text{compl}(n, \to, \mathcal{T}_0) = \sup\{\text{dh}(t, \to) \mid t \in \mathcal{T}_0, |t| \le n\}$. *The* derivational complexity *function* $\text{dc}_{\mathcal{R}/\mathcal{S}}$ *maps any* $n \in \mathbb{N}$ *to the length of the longest sequence of* $\to_{\mathcal{R}/\mathcal{S}}$*-steps starting with a term whose size is at most* n, *i.e.,* $\text{dc}_{\mathcal{R}/\mathcal{S}}(n) = \text{compl}(n, \to_{\mathcal{R}/\mathcal{S}}, \mathcal{T}^{\mathcal{R}/\mathcal{S}})$. *The* runtime complexity *function* $\text{rc}_{\mathcal{R}/\mathcal{S}}$ *is defined analogously for sequences starting with* basic *terms, i.e.,* $\text{rc}_{\mathcal{R}/\mathcal{S}}(n) = \text{compl}(n, \to_{\mathcal{R}/\mathcal{S}}, \mathcal{T}_{\text{basic}}^{\mathcal{R}/\mathcal{S}})$. *The* innermost derivational complexity *function* $\text{idc}_{\mathcal{R}/\mathcal{S}}$ *and the* innermost runtime complexity *function* $\text{irc}_{\mathcal{R}/\mathcal{S}}$ *are defined analogously, using* $\xrightarrow{\text{i}}_{\mathcal{R}/\mathcal{S}}$ *instead of* $\to_{\mathcal{R}/\mathcal{S}}$ *in the definitions.*

The new transformation preserves and reflects derivation height *precisely*. However, many complexity analysis techniques for rewriting consider *asymptotic* behavior.

▶ **Definition 3** (Asymptotic notation, $\mathcal{O}$, Ω, Θ). *Let* $f, g : \mathbb{N} \to \mathbb{N} \cup \{\omega\}$. *Then* $f(n) \in \mathcal{O}(g(n))$ *iff there are constants* $M, N \in \mathbb{N}$ *such that* $f(n) \le M \cdot g(n)$ *for all* $n \ge N$. *Moreover,* $f(n) \in \Omega(g(n))$ *iff* $g(n) \in \mathcal{O}(f(n))$, *and* $f(n) \in \Theta(g(n))$ *iff* $f(n) \in \mathcal{O}(g(n))$ *and* $f(n) \in \Omega(g(n))$.

▶ **Example 4** (plus). Consider the relative TRS $\mathcal{R}/\mathcal{S}$ with $\mathcal{R} = \{\mathsf{plus}(\mathsf{0}, x) \to x, \mathsf{plus}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{plus}(x, y))\}$ and $\mathcal{S} = \emptyset$. Here $\mathsf{0}$ and s are constructor symbols, and plus is a defined symbol. We have $\text{rc}_{\mathcal{R}/\mathcal{S}}(n) \in \Theta(n)$, $\text{irc}_{\mathcal{R}/\mathcal{S}}(n) \in \Theta(n)$, $\text{dc}_{\mathcal{R}/\mathcal{S}}(n) \in \Theta(n^2)$, and $\text{idc}_{\mathcal{R}/\mathcal{S}}(n) \in \Theta(n^2)$.

2 Transforming Derivational Complexity to Runtime Complexity

This section describes the main contribution, an instrumentation of a relative TRS $\mathcal{R}/\mathcal{S}$ to a relative TRS $\mathcal{R}/(\mathcal{S} \uplus \mathcal{G})$ with $\text{dc}_{\mathcal{R}/\mathcal{S}}(n) = \text{rc}_{\mathcal{R}/(\mathcal{S}\uplus\mathcal{G})}(n)$ and $\text{idc}_{\mathcal{R}/\mathcal{S}}(n) = \text{irc}_{\mathcal{R}/(\mathcal{S}\uplus\mathcal{G})}(n)$. The idea is to encode the set of *all* start terms (over a given signature) that must be considered for derivational complexity into a set of *basic* terms (over an *extended* signature) of the same term size that can be analyzed for runtime complexity. We add constructor symbols c_f that represent the defined symbols f from $\mathcal{R}/\mathcal{S}$ in a basic term; Thatte [12] uses a similar representation to transform arbitrary TRSs to constructor systems. We also add relative rewrite rules $\mathcal{G}$ to generate the original start term for $\mathcal{R}/\mathcal{S}$ from its encoding as a basic term for $\mathcal{R}/(\mathcal{S} \uplus \mathcal{G})$. The root symbol for these basic terms is called enc_f for a symbol f. Thus, in contrast to Thatte, we transform the start term rather than the original rules of the TRS.

▶ **Example 5** (Ex. 4 continued). A start term $\mathsf{plus}(\mathsf{plus}(\mathsf{s}(0),0),0)$ for dc will be represented by a basic term $\mathsf{enc}_{\mathsf{plus}}(\mathsf{c}_{\mathsf{plus}}(\mathsf{s}(0),0),0)$. Here $\mathsf{enc}_{\mathsf{plus}}$ will be a defined symbol and $\mathsf{c}_{\mathsf{plus}}$ a constructor symbol. Rewriting using $\xrightarrow{\mathsf{i}}_{\mathcal{G}}$ can then restore the original start term.

▶ **Definition 6** (Generator rules $\mathcal{G}$, runtime instrumentation). *Let $\mathcal{R}/\mathcal{S}$ be a relative TRS. We define the* generator rules $\mathcal{G}$ *of $\mathcal{R}/\mathcal{S}$ as the set of rules*

$$\begin{aligned}\mathcal{G} = {} & \{\mathsf{enc}_f(x_1,\dots,x_n) \to f(\mathsf{eArg}(x_1),\dots,\mathsf{eArg}(x_n)) \mid f \in \Sigma^{\mathcal{R}\cup\mathcal{S}}\}\\ & \cup \{\mathsf{eArg}(\mathsf{c}_f(x_1,\dots,x_n)) \to f(\mathsf{eArg}(x_1),\dots,\mathsf{eArg}(x_n)) \mid f \in \Sigma_d^{\mathcal{R}\cup\mathcal{S}}\}\\ & \cup \{\mathsf{eArg}(f(x_1,\dots,x_n)) \to f(\mathsf{eArg}(x_1),\dots,\mathsf{eArg}(x_n)) \mid f \in \Sigma_c^{\mathcal{R}\cup\mathcal{S}}\}\end{aligned}$$

where $x_1,\dots,x_n$ are variables and all function symbols eArg, c_f, and enc_f are fresh (i.e., do not occur in $\mathcal{R}\cup\mathcal{S}$). The relative TRS $\mathcal{R}/(\mathcal{S}\uplus\mathcal{G})$ is the runtime instrumentation *of $\mathcal{R}/\mathcal{S}$, with extended signature $\Sigma^{\mathcal{R}\cup\mathcal{S}\cup\mathcal{G}} = \Sigma^{\mathcal{R}\cup\mathcal{S}} \cup \{\mathsf{eArg}\} \cup \{\mathsf{c}_f \mid f \in \Sigma_d^{\mathcal{R}\cup\mathcal{S}}\} \cup \{\mathsf{enc}_f \mid f \in \Sigma^{\mathcal{R}\cup\mathcal{S}}\}$.*

▶ **Example 7** (Ex. 4 and Ex. 5 cont'd). For Ex. 4, we get the following generator rules $\mathcal{G}$:

$$\begin{array}{rcl|rcl}
\mathsf{enc}_{\mathsf{plus}}(x_1,x_2) & \to & \mathsf{plus}(\mathsf{eArg}(x_1),\mathsf{eArg}(x_2)) & \mathsf{eArg}(\mathsf{c}_{\mathsf{plus}}(x_1,x_2)) & \to & \mathsf{plus}(\mathsf{eArg}(x_1),\mathsf{eArg}(x_2))\\
\mathsf{enc}_0 & \to & 0 & \mathsf{eArg}(0) & \to & 0\\
\mathsf{enc}_{\mathsf{s}}(x_1) & \to & \mathsf{s}(\mathsf{eArg}(x_1)) & \mathsf{eArg}(\mathsf{s}(x_1)) & \to & \mathsf{s}(\mathsf{eArg}(x_1))
\end{array}$$

▶ **Theorem 8** ((Innermost) derivational complexity via (innermost) runtime complexity). *Let $\mathcal{R}/\mathcal{S}$ be a relative TRS and let $\mathcal{R}/(\mathcal{S}\uplus\mathcal{G})$ be its runtime instrumentation. For all $n\in\mathbb{N}$, we then have (1) $\mathrm{dc}_{\mathcal{R}/\mathcal{S}}(n) = \mathrm{rc}_{\mathcal{R}/(\mathcal{S}\uplus\mathcal{G})}(n)$, and (2) $\mathrm{idc}_{\mathcal{R}/\mathcal{S}}(n) = \mathrm{irc}_{\mathcal{R}/(\mathcal{S}\uplus\mathcal{G})}(n)$.*

▶ **Example 9** (`Derivational_Complexity_Full_Rewriting/AG01/#3.12`, TPDB [13]). As an example that was (to my knowledge) beyond automated analysis tools for derivational complexity before, but can now be handled automatically, consider the rewrite rules $\mathcal{R}$:

$$\begin{array}{rcl|rcl}
\mathsf{app}(\mathsf{nil},y) & \to & y & \mathsf{app}(\mathsf{add}(n,x),y) & \to & \mathsf{add}(n,\mathsf{app}(x,y))\\
\mathsf{reverse}(\mathsf{nil}) & \to & \mathsf{nil} & \mathsf{reverse}(\mathsf{add}(n,x)) & \to & \mathsf{app}(\mathsf{reverse}(x),\mathsf{add}(n,\mathsf{nil}))\\
\mathsf{shuffle}(\mathsf{nil}) & \to & \mathsf{nil} & \mathsf{shuffle}(\mathsf{add}(n,x)) & \to & \mathsf{add}(n,\mathsf{shuffle}(\mathsf{reverse}(x)))
\end{array}$$

The implementation of the new transformation in the tool AProVE [7] adds the following generator rules $\mathcal{G}$:

$$\begin{array}{rcl|rcl}
\mathsf{enc}_{\mathsf{nil}} & \to & \mathsf{nil} & \mathsf{eArg}(\mathsf{nil}) & \to & \mathsf{nil}\\
\mathsf{enc}_{\mathsf{add}}(x_1,x_2) & \to & \mathsf{add}(\mathsf{eArg}(x_1),\mathsf{eArg}(x_2)) & \mathsf{eArg}(\mathsf{add}(x_1,x_2)) & \to & \mathsf{add}(\mathsf{eArg}(x_1),\mathsf{eArg}(x_2))\\
\mathsf{enc}_{\mathsf{app}}(x_1,x_2) & \to & \mathsf{app}(\mathsf{eArg}(x_1),\mathsf{eArg}(x_2)) & \mathsf{eArg}(\mathsf{c}_{\mathsf{app}}(x_1,x_2)) & \to & \mathsf{app}(\mathsf{eArg}(x_1),\mathsf{eArg}(x_2))\\
\mathsf{enc}_{\mathsf{reverse}}(x_1) & \to & \mathsf{reverse}(\mathsf{eArg}(x_1)) & \mathsf{eArg}(\mathsf{c}_{\mathsf{reverse}}(x_1)) & \to & \mathsf{reverse}(\mathsf{eArg}(x_1))\\
\mathsf{enc}_{\mathsf{shuffle}}(x_1) & \to & \mathsf{shuffle}(\mathsf{eArg}(x_1)) & \mathsf{eArg}(\mathsf{c}_{\mathsf{shuffle}}(x_1)) & \to & \mathsf{shuffle}(\mathsf{eArg}(x_1))
\end{array}$$

Then AProVE determines $\mathrm{dc}_{\mathcal{R}/\emptyset}(n)\in\mathcal{O}(n^4)$ and $\mathrm{dc}_{\mathcal{R}/\emptyset}(n)\in\Omega(n^3)$. (A manual analysis shows $\mathrm{dc}_{\mathcal{R}/\emptyset}(n)\in\Theta(n^4)$.) The upper bound is found as follows: First a sufficient criterion [4] shows that for the TRS $\mathcal{R}/\mathcal{G}$, rc and irc coincide. To analyze irc, the approach by Naaf et al. [10] is applied. It encodes the search for upper bounds for irc to the search for upper time complexity bounds for integer transition systems. The proof is completed using the tools CoFloCo [3] and KoAT [2] as backends for complexity analysis of integer transition systems. The lower bound is found using rewrite lemmas [5].

Tool	$\mathcal{O}(1)$	$\leq \mathcal{O}(n)$	$\leq \mathcal{O}(n^2)$	$\leq \mathcal{O}(n^3)$	$\leq \mathcal{O}(n^{\geq 4})$
TcT direct idc	1	368	468	481	501
TcT instrumentation irc	3	465	555	626	691
AProVE instrumentation irc	13	598	769	827	833

Table 1 Upper bounds for derivational complexity of innermost rewriting

Tool	$\mathcal{O}(1)$	$\leq \mathcal{O}(n)$	$\leq \mathcal{O}(n^2)$	$\leq \mathcal{O}(n^3)$	$\leq \mathcal{O}(n^{\geq 4})$
TcT direct dc	1	366	466	479	499
TcT instrumentation rc	1	203	224	304	304
AProVE instrumentation rc	1	328	386	398	399

Table 2 Upper bounds for derivational complexity of full rewriting

3 Implementation and Experimental Evaluation

I implemented my transformation in the termination and complexity analysis tool AProVE [7]. First the runtime instrumentation of the derivational complexity problems is computed, and then this generated problem is analyzed by existing techniques to infer bounds for the runtime complexity. The configurations for innermost and full rewriting are labeled "AProVE instrumentation irc" and "AProVE instrumentation rc" in Tables 1 and 2.

I compared with the state-of-the-art complexity analysis tool TcT [1] from the Termination and Complexity Competition in 2018[1][2] to analyze derivational complexity for innermost and full rewriting, "TcT direct idc" and "TcT direct dc" in Tables 1 and 2. To assess if the new technique could be useful for existing state-of-the-art tools like TcT for derivational complexity, I extracted the runtime instrumentations for the derivational complexity benchmarks and ran experiments with TcT on the resulting runtime complexity inputs ("TcT instrumentation ...").

The experiments were run on the StarExec compute cluster [11] in the `all.q` queue with 300 seconds timeout per example. The benchmark set was based on the derivational complexity families of version 10.6 of the TPDB: 2664 benchmarks for innermost rewriting from family `Derivational_Complexity_Innermost_Rewriting` and 1754 benchmarks for full rewriting from family `Derivational_Complexity_Full_Rewriting`). I used only those benchmarks whose rewrite rules $\ell \to r$ satisfy the conditions from Def. 1 that $\ell \notin \mathcal{V}$ and $\mathcal{V}(r) \subseteq \mathcal{V}(\ell)$. Version 10.6 of the TPDB has 60 further derivational complexity examples for innermost rewriting and 55 further examples for full rewriting that are not compatible with these requirements.

Tables 1 and 2 give an overview over my experimental results for upper bounds.[3] For each configuration, I state the number of examples for which the corresponding asymptotic upper complexity bound was inferred. An entry in a row "$\leq \mathcal{O}(n^k)$" means that the tool proved a bound $\leq \mathcal{O}(n^k)$ (e.g., in Table 1, "TcT direct idc" proved constant or linear upper bounds in 368 cases).

[1] Available at: `https://www.starexec.org/starexec/secure/details/solver.jsp?id=20651`
[2] In 2019, no tools had been submitted for derivational complexity.
[3] I ran experiments for lower bounds as well [6], omitted here for space reasons.

For innermost rewriting, Table 1 shows that both TcT and AProVE benefit significantly from the new instrumentation. For example, for constant or linear upper bounds, the 2018 version of TcT inferred such bounds for 368 TRSs, but with the instrumentation, TcT found such bounds for 465 TRSs, and AProVE found such bounds for 598 TRSs.

For full rewriting, Table 2 shows that the 2018 version of TcT scores noticeably better than the instrumentation-based approach. Still, Ex. 9 shows that also here bounds on derivational complexity can now be found that were out of reach before.

The stronger impact of the transformation on innermost rewriting over full rewriting is likely due to the following reasons: (1) Not many dedicated techniques for finding upper bounds of derivational complexity of *innermost* rewriting seem to be available beyond techniques that are restricted to upper bounds of derivational complexity of full rewriting. (2) For the runtime complexity backends, stronger techniques are available for innermost rewriting than for full rewriting.

Independently of the above evaluation, also the results of TermComp 2020[4] indicate very good results for AProVE with the transformation for the derivational complexity categories. Thus, the new instrumentation-based approach is a useful addition to state-of-the-art techniques for analysis of (innermost) derivational complexity. My experimental data are available here: `http://www.dcs.bbk.ac.uk/~carsten/eval/rcdc/`

4 Extensions over the Conference Version [6]

We have encoded the set of all terms as start terms to corresponding basic terms whose derivation height corresponds to that of the original terms. This encoding lets analysis tools for runtime complexity obtain results for derivational complexity. However, the approach of introducing generator rules to create "intended start terms" is not restricted to the set of *all* terms from the original term universe. Given a relative TRS $\mathcal{R}/\mathcal{S}$, one can encode *any set* of start terms $\mathcal{T}_0$ for which there is a set $\mathcal{G}$ of rewrite rules over an extended signature such that

1. $\mathcal{T}_{\text{basic}}^{\mathcal{R}/(\mathcal{S}\uplus\mathcal{G})} \subseteq \mathcal{T}_0$ (all basic terms w.r.t. $\mathcal{R}/(\mathcal{S}\uplus\mathcal{G})$ are included), and
2. $\mathcal{T}_{\text{basic}}^{\mathcal{R}/(\mathcal{S}\uplus\mathcal{G})} \to_{\mathcal{G}}^* \mathcal{T}_0$ (in the sense of $\exists t \in \mathcal{T}_{\text{basic}}^{\mathcal{R}/(\mathcal{S}\uplus\mathcal{G})}.\ \exists t' \in \mathcal{T}_0.\ t \to_{\mathcal{G}}^* t'$) for a set of suitable relative rewrite rules $\mathcal{G}$ (all terms reachable from basic terms w.r.t. $\mathcal{R}/(\mathcal{S}\uplus\mathcal{G})$ are included).

5 Conclusion

This extended abstract sketches a new transformation to analyze derivational complexity problems in term rewriting via an off-the-shelf analysis tool for the analysis of runtime complexity. Extensive experiments validate the practical usefulness of this approach.

I recommend that a complexity analysis tool should use this approach and existing techniques for derivational complexity in parallel. For complexity analysis tools specialized to (innermost) runtime complexity, the new transformation can provide an avenue to broadened applicability. In general, the approach of using instrumentations by rewrite rules to generate the set of "intended" start terms from their representation via "allowed" start terms appears to be underexplored in the analysis of properties of rewrite systems. I believe that this approach is worth investigating further, also for other properties of rewriting.

Acknowledgments. I thank Florian Frohn and Jürgen Giesl for valuable discussions and Aart Middeldorp for the pointer to [12].

[4] `https://termcomp.github.io/Y2020/`

References

1 Martin Avanzini, Georg Moser, and Michael Schaper. TcT: Tyrolean complexity tool. In *Proc. TACAS '16*, volume 9636 of *LNCS*, pages 407–423, 2016.

2 Marc Brockschmidt, Fabian Emmes, Stephan Falke, Carsten Fuhs, and Jürgen Giesl. Analyzing runtime and size complexity of integer programs. *ACM TOPLAS*, 38(4):13:1–13:50, 2016.

3 Antonio Flores-Montoya and Reiner Hähnle. Resource analysis of complex programs with cost equations. In *Proc. APLAS '14*, volume 8858 of *LNCS*, pages 275–295, 2014.

4 Florian Frohn and Jürgen Giesl. Analyzing runtime complexity via innermost runtime complexity. In *Proc. LPAR '17*, volume 46 of *EPiC*, pages 249–268, 2017.

5 Florian Frohn, Jürgen Giesl, Jera Hensel, Cornelius Aschermann, and Thomas Ströder. Lower bounds for runtime complexity of term rewriting. *Journal of Automated Reasoning*, 59(1):121–163, 2017.

6 Carsten Fuhs. Transforming derivational complexity of term rewriting to runtime complexity. In *Proc. FroCoS '19*, volume 11715 of *LNAI*, pages 348–364, 2019.

7 Jürgen Giesl, Cornelius Aschermann, Marc Brockschmidt, Fabian Emmes, Florian Frohn, Carsten Fuhs, Jera Hensel, Carsten Otto, Martin Plücker, Peter Schneider-Kamp, Thomas Ströder, Stephanie Swiderski, and René Thiemann. Analyzing program termination and complexity automatically with AProVE. *Journal of Automated Reasoning*, 58:3–31, 2017.

8 Nao Hirokawa and Georg Moser. Automated complexity analysis based on the dependency pair method. In *Proc. IJCAR '08*, volume 5195 of *LNAI*, pages 364–379, 2008.

9 Dieter Hofbauer and Clemens Lautemann. Termination proofs and the length of derivations. In *Proc. RTA '89*, volume 355 of *LNCS*, pages 167–177, 1989.

10 Matthias Naaf, Florian Frohn, Marc Brockschmidt, Carsten Fuhs, and Jürgen Giesl. Complexity analysis for term rewriting by integer transition systems. In *Proc. FroCoS '17*, volume 10483 of *LNAI*, pages 132–150, 2017.

11 Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. Starexec: A cross-community infrastructure for logic solving. In *Proc. IJCAR '14*, volume 8562 of *LNAI*, pages 367–373, 2014.

12 Satish Thatte. Implementing first-order rewriting with constructor systems. *Theoretical Computer Science*, 61(1):83–92, 1988.

13 Wiki. Termination Problems DataBase. `http://termination-portal.org/wiki/TPDB`.

14 Harald Zankl and Martin Korp. Modular complexity analysis for term rewriting. *Logical Methods in Computer Science*, 10(1), 2014.

Mixed Base Rewriting for the Collatz Conjecture

Emre Yolcu
Carnegie Mellon University, Pittsburgh, PA 15213, USA

Scott Aaronson
University of Texas at Austin, Austin, TX 78712, USA

Marijn J. H. Heule
Carnegie Mellon University, Pittsburgh, PA 15213, USA

Abstract

We explore the Collatz conjecture and its variants through the lens of termination of string rewriting. We construct a rewriting system that simulates the iterated application of the Collatz function on strings corresponding to mixed binary–ternary representations of positive integers. We prove that the termination of this rewriting system is equivalent to the Collatz conjecture. We also prove that a previously studied rewriting system that simulates the Collatz function using unary representations does not admit termination proofs via matrix interpretations. To show the feasibility of our approach in proving mathematically interesting statements, we implement a minimal termination prover that uses matrix/arctic interpretations and we find automated proofs of nontrivial weakenings of the Collatz conjecture. Although we do not succeed in proving the Collatz conjecture, we believe that the ideas here represent an interesting new approach.

2012 ACM Subject Classification Theory of computation → Automated reasoning; Theory of computation → Rewrite systems

Keywords and phrases string rewriting, termination, matrix interpretations, SAT solving, Collatz conjecture, computer-assisted mathematics

Related Version This work is a short version of a paper appearing at CADE-28.
Extended preprint: `https://arxiv.org/abs/2105.14697`

Supplementary Material *Code*: `https://github.com/emreyolcu/rewriting-collatz`

1 Introduction

Let $\mathbb{N}$ and $\mathbb{N}^+$ denote the natural numbers and the positive integers, respectively. We define the *Collatz function* $T\colon \mathbb{N}^+ \to \mathbb{N}^+$ as $T(n) = n/2$ if $n \equiv 0 \pmod 2$ and $T(n) = (3n+1)/2$ if $n \equiv 1 \pmod 2$. With T^k denoting the kth iterate of T, the well-known *Collatz conjecture* [4] states that for all $n \in \mathbb{N}^+$, there exists some $k \in \mathbb{N}$ such that $T^k(n) = 1$. More generally, letting X be either $\mathbb{N}$ or $\mathbb{N}^+$, we consider a function $f\colon X \to X$. We call the sequence $x, f(x), f^2(x), \ldots$ the *f-trajectory of x*. If this trajectory contains 1, it is called *convergent*. If all f-trajectories are convergent, we say that f is *convergent*.

In this paper, we describe an approach based on termination of string rewriting to automatically search for a proof of the Collatz conjecture. Although trying to prove the Collatz conjecture via automated deduction is clearly a moonshot goal, there are two technological advances that provide reasons for optimism that at least some interesting variants of the problem might be solvable. First, the invention of the method of matrix interpretations [1] and its variants such as arctic interpretations [3] turns the quest of finding a ranking function to witness termination into a problem that is suitable for systematic search. Second, the progress in satisfiability (SAT) solving makes it possible to solve many seemingly difficult combinatorial problems efficiently in practice. Their combination, i.e., using SAT solvers to find interpretations, has so far been effective in solving challenging termination problems. We make the following contributions:

- We show how a Collatz-like function can be expressed as a rewriting system that is terminating if and only if the function is convergent.
- We prove that no termination proof via matrix interpretations exists for a certain system that simulates the Collatz function using unary representations of numbers.
- We show that translations into rewriting systems that use non-unary representations of numbers are more amenable to automated methods, compared with the previously and commonly studied unary representations.
- We automatically prove various weakenings of the Collatz conjecture. We observe that, for some of these weakenings, the only matrix/arctic interpretations that our termination tool was able to find involved relatively large matrices (of dimension 5). Existing termination tools often limit their default strategies to search for small interpretations as they are tailored for the setting where the task is to quickly solve a large quantity of relatively easy problems. We make the point that, given more resources, the method of matrix/arctic interpretations has the potential to scale.

2 Rewriting the Collatz Function

We start with systems that use unary representations and then demonstrate via examples that mixed base representations can be more suitable for use with automated methods.

Rewriting in Unary. The following system of Zantema [5] simulates the iterated application of the Collatz function to a number represented in unary, and it terminates upon reaching 1.

▶ **Example 1.** $\mathcal{Z}$ denotes the following SRS, consisting of 5 symbols and 7 rules.

$$\begin{array}{lll} \mathtt{h11} \to \mathtt{1h} & \mathtt{11h}\diamond \to \mathtt{11s}\diamond & \mathtt{h1}\diamond \to \mathtt{t11}\diamond \\ & \mathtt{1s} \to \mathtt{s1} & \mathtt{1t} \to \mathtt{t111} \\ & \diamond\mathtt{s} \to \diamond\mathtt{h} & \diamond\mathtt{t} \to \diamond\mathtt{h} \end{array}$$

▶ **Theorem 2** ([5, Theorem 16]). *$\mathcal{Z}$ is terminating if and only if the Collatz conjecture holds.*

While the forward direction of the above theorem is easy to see (since $\diamond\mathtt{h1}^{2n}\diamond \to^*_{\mathcal{Z}} \diamond\mathtt{h1}^{n}\diamond$ for $n > 1$ and $\diamond\mathtt{h1}^{2n+1}\diamond \to^*_{\mathcal{Z}} \diamond\mathtt{h1}^{3n+2}\diamond$ for $n \geq 0$), the backward direction is far from obvious because not every string corresponds to a valid configuration of the underlying machine.

As another example, consider the system $\mathcal{W} = \{\mathtt{h11} \to \mathtt{1h}, \mathtt{1h}\diamond \to \mathtt{1t}\diamond, \mathtt{1t} \to \mathtt{t111}, \diamond\mathtt{t} \to \diamond\mathtt{h}\}$ (originally due to Zantema, available at: `https://www.lri.fr/~marche/tpdb/tpdb-2.0/SRS/Zantema/z079.srs`). Termination of this system has yet to be proved via automated methods. Nevertheless, there is a simple reason for its termination: It simulates the iterated application of a Collatz-like function $W\colon \mathbb{N}^+ \to \mathbb{N}^+$ defined as $W(n) = 3n/2$ if $n \equiv 0 \pmod 2$ and $W(n) = 1$ if $n \equiv 1 \pmod 2$, which is easily seen to be convergent.

Matrix interpretations cannot be used to remove any of the rules from the above kind of unary rewriting systems that simulate certain maps, in particular the Collatz function. We prove the below theorem in the full version of this work. We adopt the notation of [1].

▶ **Theorem 3.** *Let $\Sigma = \{\mathtt{1}, \diamond, \mathtt{h}, \mathtt{s}, \mathtt{t}\}$. There exists no collection $[\cdot]_\Sigma$ of matrix interpretations of any dimension d such that*

- *for at least a rule $\ell \to r \in \mathcal{Z}$ we have $[\ell](\mathbf{x}) > [r](\mathbf{x})$ for all $\mathbf{x} \in \mathbb{N}^d$, and*
- *for the remaining $\ell' \to r' \in \mathcal{Z}$ we have $[\ell'](\mathbf{x}) \gtrsim [r'](\mathbf{x})$ for all $\mathbf{x} \in \mathbb{N}^d$.*

By an argument analogous to above, we can also prove that no such collection of interpretations exists for $\mathcal{W}$. If a proof of the Collatz conjecture is to be produced by some automated method that relies on rewriting, then that method better be able to prove a statement as simple as the convergence of W. With this in mind, we describe an alternative rewriting system that simulates the Collatz function and terminates upon reaching 1. We then provide examples where the alternative system is more suitable for use with termination tools (for instance allowing a matrix interpretations proof of the convergence of W).

Rewriting in Mixed Base. In the mixed base scheme, the overall idea is as follows. Given a number $n \in \mathbb{N}^+$, we write a mixed binary–ternary representation for it (noting that this representation is not unique). With this representation, as long as the least significant digit is binary, the parity of the number can be recognized by checking only this digit, as opposed to scanning the entire string when working in unary. This allows us to easily determine the correct case when applying the Collatz function. If the least significant digit is ternary, then the representation is rewritten (while preserving its value) to make this digit binary. Afterwards, since computing $2n \mapsto n$ corresponds to erasing a trailing binary 0 and computing $2n+1 \mapsto 3n+2$ corresponds to replacing a trailing binary 1 with a ternary 2, applying the Collatz function takes a single rewrite step.

We will describe an SRS $\mathcal{T}$ over the symbols $\{\mathtt{f}, \mathtt{t}, \mathtt{0}, \mathtt{1}, \mathtt{2}, \triangleleft, \triangleright\}$ that simulates the iterated application of the Collatz function and terminates upon reaching 1. The symbols $\mathtt{f}, \mathtt{t}$ correspond to binary digits $0_2, 1_2$; and $\mathtt{0}, \mathtt{1}, \mathtt{2}$ to ternary digits $0_3, 1_3, 2_3$. The symbol $\triangleleft$ marks the beginning of a string while also standing for the most significant digit (without loss of generality assumed to be 1_0) and $\triangleright$ marks the end of a string while also standing for the redundant trailing digit 0_1. Consider the functional view of these symbols:

$$\begin{array}{ll} \mathtt{f}(x) = 2x & \mathtt{0}(x) = 3x \\ \mathtt{t}(x) = 2x+1 & \mathtt{1}(x) = 3x+1 \\ & \mathtt{2}(x) = 3x+2 \end{array} \qquad \begin{array}{ll} \triangleleft(x) = 1 \\ \triangleright(x) = x \end{array} \tag{1}$$

A mixed base representation $N = (n_1)_{b_1}(n_2)_{b_2}\ldots(n_k)_{b_k}$ represents the number $\mathrm{Val}(N) \coloneqq \sum_{i=1}^{k} n_i \prod_{j=i+1}^{k} b_j$. We can see by rearranging this expression that $\mathrm{Val}(N)$ is also given by some composition of the above functions if we view the expression $\triangleleft(x)$ as the constant 1.

▶ **Example 4.** We can write $19 = \mathrm{Val}(\triangleleft\mathtt{0f1}\triangleright) = \triangleright(\mathtt{1}(\mathtt{f}(\mathtt{0}(\triangleleft(x)))))$. The string representation ends with a ternary symbol, so we will rewrite it. With the function view, we have $\mathtt{1}(\mathtt{f}(x)) = 3(2x)+1 = 6x+1 = 2(3x)+1 = \mathtt{t}(\mathtt{0}(x))$. This shows that we could also write $19 = \mathrm{Val}(\triangleleft\mathtt{00t}\triangleright)$, which now ends with the binary digit 1_2. This gives us the rewrite rule $\mathtt{f1} \to \mathtt{0t}$. We can now apply the Collatz function to this representation by rewriting only the rightmost two symbols of the string since $T(\triangleright(\mathtt{t}(x))) = \frac{3(2x+1)+1}{2} = \frac{6x+4}{2} = 3x+2 = (\triangleright(\mathtt{2}(x)))$. This gives us the rewrite rule $\mathtt{t}\triangleright \to \mathtt{2}\triangleright$. After applying this rule to the string $\triangleleft\mathtt{00t}\triangleright$, we indeed obtain $T(19) = 29 = \mathrm{Val}(\triangleleft\mathtt{002}\triangleright)$.

In the manner of the above example, we compute all the necessary transformations and obtain the following 11-rule SRS $\mathcal{T}$.

$$\mathcal{D}_T = \left\{ \begin{array}{l} \mathtt{f}\triangleright \to \triangleright \\ \mathtt{t}\triangleright \to \mathtt{2}\triangleright \end{array} \right\} \qquad \mathcal{A} = \left\{ \begin{array}{ll} \mathtt{f0} \to \mathtt{0f} & \mathtt{t0} \to \mathtt{1t} \\ \mathtt{f1} \to \mathtt{0t} & \mathtt{t1} \to \mathtt{2f} \\ \mathtt{f2} \to \mathtt{1f} & \mathtt{t2} \to \mathtt{2t} \end{array} \right\} \qquad \mathcal{B} = \left\{ \begin{array}{l} \triangleleft\mathtt{0} \to \triangleleft\mathtt{t} \\ \triangleleft\mathtt{1} \to \triangleleft\mathtt{ff} \\ \triangleleft\mathtt{2} \to \triangleleft\mathtt{ft} \end{array} \right\}$$

This SRS is split into subsystems $\mathcal{D}_T$ (dynamic rules for T) and $\mathcal{X} = \mathcal{A} \cup \mathcal{B}$ (auxiliary rules). The two rules in $\mathcal{D}_T$ encode the application of the Collatz function T, while the rules in $\mathcal{X}$

serve to push binary symbols towards the rightmost end of the string by swapping the bases of adjacent positions without changing the represented value.

▶ **Example 5** (Rewrite sequence of $\mathcal{T}$)**.** Consider the string $s = \triangleleft\mathtt{ff0}\triangleright$ that represents the number 12. Below is a possible rewrite sequence of $\mathcal{T}$ that starts from s, with the corresponding values (under the interpretations from (1)) displayed above the strings. Underlines indicate the parts of the strings where the rules are applied.

$$\begin{array}{ccccccccccccccc}
12 & & 12 & & 6 & & 6 & & 3 & & 3 & & 5 & & 5 \\
\triangleleft\mathtt{ff}\underline{\mathtt{0}}\triangleright & \to_{\mathcal{A}} & \triangleleft\mathtt{f}\underline{\mathtt{0f}}\triangleright & \to_{\mathcal{D}_T} & \triangleleft\underline{\mathtt{f0}}\triangleright & \to_{\mathcal{A}} & \triangleleft\mathtt{0}\underline{\mathtt{f}}\triangleright & \to_{\mathcal{D}_T} & \underline{\triangleleft\mathtt{0}}\triangleright & \to_{\mathcal{B}} & \triangleleft\underline{\mathtt{t}}\triangleright & \to_{\mathcal{D}_T} & \underline{\triangleleft\mathtt{2}}\triangleright & \to_{\mathcal{B}} & \triangleleft\mathtt{f}\underline{\mathtt{t}}\triangleright \\
8 & & 8 & & 8 & & 4 & & 2 & & 1 & & & & \\
\to_{\mathcal{D}_T} \triangleleft\underline{\mathtt{f2}}\triangleright & \to_{\mathcal{A}} & \underline{\triangleleft\mathtt{1}}\mathtt{f}\triangleright & \to_{\mathcal{B}} & \triangleleft\mathtt{ff}\underline{\mathtt{f}}\triangleright & \to_{\mathcal{D}_T} & \triangleleft\mathtt{f}\underline{\mathtt{f}}\triangleright & \to_{\mathcal{D}_T} & \triangleleft\underline{\mathtt{f}}\triangleright & \to_{\mathcal{D}_T} & \triangleleft\triangleright & & & &
\end{array}$$

The trajectory of T would continue upon reaching 1; however, in order to be able to formulate the Collatz conjecture as a termination problem, $\mathcal{T}$ is made in such a way that its rewrite sequences stop upon reaching the string representation $\triangleleft\triangleright$ of 1 since no rule is applicable.

Termination of the subsystems of $\mathcal{T}$ with $\mathcal{B}$ or $\mathcal{D}_T$ removed is easily seen. There is also a direct proof via linear polynomial interpretations after reversing the rules.

▶ **Lemma 6.** $\mathrm{SN}(\mathcal{T} \setminus \mathcal{B})$ *and* $\mathrm{SN}(\mathcal{T} \setminus \mathcal{D}_T)$*.*

When considering the termination of $\mathcal{T}$, it suffices to limit the discussion to initial strings of a specific form that we have been working with so far, e.g., in Examples 4 and 5.

▶ **Lemma 7.** *If $\mathcal{T}$ is terminating on all initial strings of the* canonical form $\triangleleft(\mathtt{f}|\mathtt{t}|\mathtt{0}|\mathtt{1}|\mathtt{2})^*\triangleright$*, then $\mathcal{T}$ is terminating (on all initial strings).*

As a whole, the rewriting system $\mathcal{T}$ simulates the iterated application of T (except at 1). Making use of Lemmas 6 and 7, we prove the following in the full version of this work.

▶ **Theorem 8.** *$\mathcal{T}$ is terminating if and only if T is convergent.*

3 Automated Proofs

We adapt the rewriting system $\mathcal{T}$ for different Collatz-like functions to explore the effectiveness of the mixed base scheme on weakened variants of the Collatz conjecture.

Convergence of W. Earlier we mentioned a Collatz-like function W as a simple example that could serve as a sanity check for an automated method aiming to solve Collatz-like problems. With the mixed binary–ternary scheme, this function can be seen to be simulated by the system $\mathcal{W}' = \{\mathtt{f}\triangleright \to \mathtt{0}\triangleright\} \cup \mathcal{X}$. A small matrix interpretations proof is found for this system in less than a second, in contrast to its variant $\mathcal{W}$ that uses unary representations for which no automated proof is known.

Farkas' Variant. Farkas [2] studied a slight modification of the Collatz function for which it becomes possible to prove convergence via induction. We consider automatically proving the convergence of this function as another test case for the mixed base scheme that is easier than the Collatz conjecture without being entirely trivial. Below, we define a function $F\colon \mathbb{N} \to \mathbb{N}$ that is equivalent to Farkas' definition in terms of convergence while resembling the Collatz function even more closely (with respect to the definitions of the cases). This variant is

obtained by introducing an additional case in the Collatz function for $n \equiv 1 \pmod 3$ and applying T otherwise. Its definition and a set $\mathcal{D}_F$ of dynamic rules are shown below.

$$F(n) = \begin{cases} \frac{n-1}{3} & \text{if } n \equiv 1 \pmod 3 \\ \frac{n}{2} & \text{if } n \equiv 0 \text{ or } n \equiv 2 \pmod 6 \\ \frac{3n+1}{2} & \text{if } n \equiv 3 \text{ or } n \equiv 5 \pmod 6 \end{cases} \qquad \mathcal{D}_F = \left\{ \begin{array}{l} \texttt{1}\triangleright \to \triangleright \\ \texttt{0f}\triangleright \to \texttt{0}\triangleright \\ \texttt{1f}\triangleright \to \texttt{1}\triangleright \\ \texttt{1t}\triangleright \to \texttt{12}\triangleright \\ \texttt{2t}\triangleright \to \texttt{22}\triangleright \end{array} \right\}$$

Termination of the rewriting system $\mathcal{F} = \mathcal{D}_F \cup \mathcal{X}$ is equivalent to the convergence of F. The proof of the equivalence is similar to that of Theorem 8, with the difference that when constructing a nonterminating rewrite sequence from a nonconvergent trajectory we write the first number in the trajectory in ternary (except for the most significant digit when it is a ternary 2, in which case we replace it with $\triangleleft\texttt{f}$) and always perform the rightmost possible rewrite so that a dynamic rule is applied as soon as it becomes available.

Farkas gave an inductive proof of convergence for (a variant of) F via case analysis. We found an automated proof that $\mathcal{F}$ is terminating via arctic interpretations (where the proof appears to require matrices of dimension 5 for certain steps). It is worth mentioning that the default configurations of the existing termination tools (e.g., AProVE, Matchbox) are too conservative to prove the termination of this system, but after their authors tweaked the strategies they were also able to find automated proofs via arctic interpretations.

Subsets of $\mathcal{T}$. It is also interesting to consider whether we can automatically prove the terminations of proper subsets of $\mathcal{T}$. Specifically, we considered the 11 subsystems obtained by leaving out a single rewriting rule from $\mathcal{T}$, and we found termination proofs via matrix/arctic interpretations for all of the 11 subproblems. Our interest in these problems is threefold:

1. Termination of $\mathcal{T}$ implies the terminations of all of its subsystems, so proving its termination is at least as difficult a task as proving the terminations of the 11 subsystems. Therefore, the subproblems serve as additional sanity checks that an automated approach aspiring to succeed for the Collatz conjecture ought to be able to pass.
2. Having proved the terminations of all 11 subsystems is a partial solution to the full problem, since it implies that for any single rule $\ell \to r \in \mathcal{T}$, proving that $\ell \to r$ is terminating relative to $\mathcal{T}$ settles the Collatz conjecture.
3. After the removal of a rule, the termination of the remaining system still encodes a valid mathematical question about the Collatz trajectories, i.e., the system does not become terminating for a trivial reason.

Table 1 shows the parameters of the matrix/arctic interpretations proofs that we found for the termination of each subsystem. For each rule $\ell \to r$ that is left out, we searched for a stepwise proof to show that $\mathcal{T} \setminus \{\ell \to r\}$ is terminating. On the table, we report the smallest parameters (in terms of matrix dimension) that work for all of the proof steps. In the experiments we searched (with a timeout of 30 seconds) for matrices of up to 7 dimensions, with the coefficients taking at most 8 different values.

Collatz Trajectories Modulo 8. Let m be a power of 2. Given $k \in \{0, 1, \ldots, m-1\}$, is it the case that all nonconvergent Collatz trajectories contain some $n \equiv k \pmod m$? For several values of k this can be proved to hold by inspecting the transitions of the iterates in the Collatz trajectories across residue classes modulo m. These questions can also be formulated as the terminations of some rewriting systems. With this approach we found automated proofs for several cases, which are also not difficult to prove by hand.

Table 1 Smallest proofs found for the terminations of subsystems of $\mathcal{T}$. The columns show the matrix dimension D and the maximum number V of distinct coefficients that appear in the matrices, along with the median time to find an entire proof across 25 repetitions for the fixed D and V.

	Matrix			Arctic		
Rule removed	D	V	Time	D	V	Time
`f▷` $\to$ `▷`	3	4	1.42s	3	5	15.95s
`t▷` $\to$ `2▷`	1	2	0.27s	1	3	0.28s
`f0` $\to$ `0f`	4	2	0.92s	3	4	2.46s
`f1` $\to$ `0t`	1	3	0.50s	1	4	0.51s
`f2` $\to$ `1f`	1	2	0.38s	1	3	0.39s
`t0` $\to$ `1t`	4	3	1.20s	3	4	0.87s
`t1` $\to$ `2f`	5	2	0.89s	4	3	0.84s
`t2` $\to$ `2t`	4	4	10.00s	2	5	0.62s
`◁0` $\to$ `◁t`	2	2	0.40s	2	3	0.42s
`◁1` $\to$ `◁ff`	3	3	0.53s	3	4	0.57s
`◁2` $\to$ `◁ft`	4	4	7.51s	4	3	4.04s

▶ **Theorem 9.** *If there exists a nonconvergent Collatz trajectory, it cannot avoid the residue classes of* 2, 3, 4, 6 *modulo* 8.

It remains open whether the above holds for the residue classes of 0, 1, 5, 7 modulo 8.

4 Future Work

Several extensions to this work can further our understanding of the potential of rewriting techniques for answering mathematical questions. For instance, it is of interest to study the efficacy of different termination proving techniques on the problems that we considered. We found matrix/arctic interpretations to be the most successful for our purposes despite experimenting with existing tools that implement newer techniques developed for automatically proving the terminations of a few select challenging instances. It might also be possible to prove that there exist no matrix/arctic interpretations to establish the termination of the Collatz system $\mathcal{T}$. This would be an interesting result in itself. Another issue is the matter of representation; specifically, it is worth exploring whether there exists a suitable translation of the Collatz conjecture into the termination of a term, instead of string, rewriting system since many automated termination proving techniques are generalized to term rewriting. Finally, injecting problem-specific knowledge into the rewriting systems or the termination techniques would be helpful as there exists a wealth of information about the Collatz conjecture that could simplify the search for a termination proof.

References

1 Jörg Endrullis, Johannes Waldmann, and Hans Zantema. Matrix interpretations for proving termination of term rewriting. *Journal of Automated Reasoning*, 40(2):195–220, 2008.

2 Hershel M. Farkas. Variants of the $3N + 1$ conjecture and multiplicative semigroups. In *Geometry, Spectral Theory, Groups, and Dynamics*, pages 121–127. 2005.

3 Adam Koprowski and Johannes Waldmann. Max/plus tree automata for termination of term rewriting. *Acta Cybernetica*, 19(2):357–392, 2009.

4 Jeffrey C. Lagarias. *The Ultimate Challenge: The* $3x + 1$ *Problem*. American Mathematical Society, 2010.

5 Hans Zantema. Termination of string rewriting proved automatically. *Journal of Automated Reasoning*, 34(2):105–139, 2005.

Formalizing Higher-Order Termination in Coq

Deivid Vale
Institute for Computation and Information Sciences, Radboud University, The Netherlands

Niels van der Weide
Institute for Computation and Information Sciences, Radboud University, The Netherlands

Abstract

We describe a formalization of higher-order rewriting theory and formally prove that an AFS is strongly normalizing if it can be interpreted in a well-founded domain. To do so, we use Coq, which is a proof assistant based on dependent type theory. Using this formalization, one can implement several termination techniques, like the interpretation method or dependency pairs, and prove their correctness. Those implementations can then be extracted to OCaml, which results in a verified termination checker.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Equational logic and rewriting

Keywords and phrases higher-order rewriting, Coq, termination, formalization

Funding *Deivid Vale*: Author supported by NWO project "ICHOR", NWO 612.001.803/7571.

1 Introduction

Termination, while crucial for software correctness, is difficult to check in practice. For this reason, various tools have been developed that can automatically check termination of a given program. Furthermore, such tools are often based on first-order rewriting, for example, AProVE and NaTT. However, whereas termination techniques evolved and became more sophisticated over the years, termination checkers have become more complicated as a result. Since the proofs outputted by these tools tend to be large, it is difficult for humans to check for correctness.

For termination tools based on higher-order rewriting [8], this is even more the case. Higher-order rewriting is an extension of first-order rewriting in which function symbols might also have arbitrary functions as argument. Such an extension gives extra expressiveness, because higher-order rewriting allows one to deal with higher-order functional programs. It also lies at the backend of various theorem provers, it can be used for code transformation in compilers, and more. However, with greater expressiveness comes a more difficult theory and more elaborated tools that check for termination are needed. This leads to the following problem: how can we formally guarantee the correctness of the answers given by such tools?

This problem already got quite some attention. One approach is not to prove that the termination checker is correct, but instead, to check the correctness of the certificates it outputs. Such an approach was taken by Contejean *et al.* [4] and in CoLoR, developed by Blanqui, Koprowski, and others [2]. Those tools take as input a certificate produced by a termination checker, and then the proof assistant Coq checks whether it can reconstruct a termination proof. If Coq says yes, then the result was actually correct. Note that Contejean *et al.* deal with the first order case, while CoLoR also includes some methods applicable to higher-order rewriting.

In Isabelle [14], the library IsaFoR contains numerous results about first order rewriting. Such results can be used to automatically verify the termination of functions defined in Isabelle using a tool that produces certificates of termination [9]. In addition, algorithmic methods were implemented using this library, and from that, they extracted the verified termination checker called CeTA [13]. More recent efforts have also been put in formalizing

other tools, for instance, Thiemann and Sternagel developed a verified tool for certifying AProVE's termination proofs of LLVM IR programs [13, 6].

Our goal is to develop a verified termination checker for higher-order rewrite systems. To do so, we start by formalizing basic theory on rewriting in the proof assistant Coq [1]. After that, we can implement several algorithms, such as the interpretation method, dependency pairs, path orders, and use the theory to prove their correctness. With all of that in place, we can use extraction to obtain an OCaml implementation that satisfies the given specifications [10]. Note that the extraction mechanism of Coq was proven to be correct using MetaCoq [12].

In this paper, we discuss a formalization of the basic theory of higher-order rewriting. To do so, we start by discussing signatures for algebraic functional systems in Section 2. The definitions introduced there are the basic data types of the tool. In Section 3, we discuss the main theorem that guarantees the correctness of semantical methods. We conclude with an overview of what we plan to do in Section 4.

Formalization. All definitions and theorems in this paper have been formalized with the Coq proof assistant [1]. The formalization is available at `https://github.com/nmvdw/Nijn`. Links to relevant definitions in the code are highlighted as dashed boxes.

2 Higher-Order Rewriting

In this work, we consider *Algebraic Functional Systems* (AFSs), a slightly simplified form of a higher-order functional language introduced by Jouannaud and Okada [7]. This choice gives an easy presentation as it combines algebraic definitions in a first-order style with a functional mechanism using λ-abstractions and term applications. It is also the higher-order format used in the Termination Competition. This gives us access to a variety of higher-order systems defined in the *termination problems database* [3], which we plan to use as benchmark for our tool in the future.

In this section, we define the notion of AFS, and we describe how we formalized this notion in Coq. The definitions given here correspond to those usually given in the literature [5] while the definitions in Coq deviate slightly. Note that since we want to extract our formalization to an OCaml program, we use a *deep embedding*, which is similar to the approach taken in CeTA and CoLoR [2, 13]. This means that for all relevant notions, such as terms and algebraic functional systems, we define types that represent them. On the contrast, one could also use a *shallow embedding* where the relevant operations are represented as functions [4].

Before we can say what an AFS consists of, we need to define types, well-typed terms, and rewrite rules. We start by looking at types.

▶ **Definition 2.1** (Types)**.** Given a set $\mathcal{B}$ of base types. The set $\mathcal{ST}_{\mathcal{B}}$ of **simple types** is inductively built from $\mathcal{B}$ using the right-associative type constructor $\Rightarrow$. Formally,

```
Inductive ty (B : Type) : Type :=
| Base : B → ty B
| Fun : ty B → ty B → ty B.
```

In the formalization, we use `A1 ⟶ A2` to denote function types. The next step is the notion of well-typed term, which are terms together with a typing derivation. This is where the *pen-and-paper* presentation deviates from the formalization. While we present named terms, the formalization, however, makes use of *De Bruijn indices*, so variables are *nameless*.

This choice affects how to formalize variable environments and terms. Notwithstanding, the two presentations are equivalent. The former gives a more pleasant informal presentation and the latter makes the formalization easier. Note that in the formalization, we use the terminology "context" instead of variable environment: that is because in type theory, variable environments usually are called contexts.

▶ **Definition 2.2** (Var Env.)**.** A **variable environment** Γ is a finite list of variable type declarations of the form $x : A$ where the variables x are pairwise distinct. Formally,

```
Inductive con (B : Type) : Type :=
| Empty : con B
| Extend : ty B → con B → con B.
```

In what follows we sugar `Extend A C` as `A ,, C` to denote the extension of variable environments. Now we can define the type of variables as the positions in an environment.

```
Inductive var {B : Type} : con B → ty B → Type :=
| Vz : forall (C : con B) (A : ty B),
   var (A ,, C) A
| Vs : forall (C : con B) (A1 A2 : ty B),
   var C A2 → var (A1 ,, C) A2.
```

The set of terms is generated by a set of *function symbols* and the usual constructors from the simply typed lambda calculus (abstraction, application, and variables). Since terms come together with a typing derivation, we must also know the type of each function symbol. Hence, when we define the notion of well-typed term, we must assume that we have a function `ar` that assigns to each function symbol a type.

▶ **Definition 2.3** (Terms)**.** Suppose, we have a map $\mathtt{ar} : \mathcal{F} \to \mathcal{ST}_{\mathcal{B}}$. For each type A, we define the set $\mathsf{Tm}(\Gamma, A)$ of **well-typed terms** of type A in the variable environment Γ by the following clauses:

- given a function symbol $\mathsf{f} \in \mathcal{F}$, we have a term $\mathsf{f} \in \mathsf{Tm}(\Gamma, \mathtt{ar}(\mathsf{f}))$;
- for each variable $x : A \in \Gamma$ we have a term $x \in \mathsf{Tm}(\Gamma, A)$;
- given a term $s \in \mathsf{Tm}(\Gamma \cup \{x : B\}, C)$, we get a term $\lambda x.s \in \mathsf{Tm}(\Gamma, B \Rightarrow C)$;
- given a term $s \in \mathsf{Tm}(\Gamma, B \Rightarrow C)$ and $t \in \mathsf{Tm}(\Gamma, B)$, we have $s\ t \in \mathsf{Tm}(\Gamma, C)$.

Given a variable environment Γ and a type A, we write $\Gamma \vdash s : A$ to denote that s is a term of type A in Γ.

To formalize the notion of terms, we make use of dependent types, which belong to the core features of proof assistants based on Martin-Löf Type Theory. This is because $\mathsf{Tm}(\Gamma, A)$ does not only depend on sets, but also on their inhabitants. It is formally expressed as below:

```
Inductive tm {B : Type} {F : Type} (ar : F → ty B) (C : con B) : ty B → Type :=
| BaseTm : forall (f : F),
    tm ar C (ar f)
| TmVar : forall (A : ty B),
    var C A → tm ar C A
| Lam : forall (A1 A2 : ty B),
    tm ar (A1 ,, C) A2 → tm ar C (A1 ⟶ A2)
| App : forall (A1 A2 : ty B),
    tm ar C (A1 ⟶ A2) → tm ar C A1 → tm ar C A2.
```

Substitutions play an important role in the formalism, we use them to instantiate rewrite rules and the rewriting relation.

▶ **Definition 2.4** (Substitution)**.** A **substitution** γ is a finite type-preserving map from variables to terms. The application of γ to s is denoted by $s\gamma$.

▶ **Definition 2.5** (Rewrite rule)**.** A **rewriting rule** is a pair of terms $\ell \to r$ of the same type. Given a set of rewriting rules $\mathcal{R}$, the **rewrite relation** induced by $\mathcal{R}$ on the set of terms is the smallest monotonic relation that is stable under substitution and contains both all elements of $\mathcal{R}$ and β-reduction. That is, it is inductively generated by:

$$\begin{array}{rcll} \ell\gamma & \to_{\mathcal{R}} & r\gamma & \text{if } \ell \to r \in \mathcal{R} \\ s\ u & \to_{\mathcal{R}} & t\ u & \text{if } s \to_{\mathcal{R}} t \\ (\lambda x.s)\ t & \to_{\mathcal{R}} & s[x := t] & \end{array} \qquad \begin{array}{rcll} u\ s & \to_{\mathcal{R}} & u\ t & \text{if } s \to_{\mathcal{R}} t \\ \lambda x.s & \to_{\mathcal{R}} & \lambda x.t & \text{if } s \to_{\mathcal{R}} t \end{array}$$

Putting all this data together, gives us the *formalized* notion of *signature for an AFS*.

▶ **Definition 2.6.** (Signature) A **signature for an AFS** consists of the following ingredients:
- a set $\mathcal{B}$ of base types and a set $\mathcal{F}$ of function symbols;
- a map $\mathtt{ar} : \mathcal{F} \to \mathcal{ST}_{\mathcal{B}}$ and a set $\mathcal{R}$ of rewriting rules.

▶ Remark (Nomenclature). It is worth mentioning that Definition 2.6 deviates from the standard notion of signature as is commonly defined in the rewriting community. There, a signature (it may be typed or not) is usually defined as a set of symbols used to generate terms. Our choice to do so differently is mainly motivated by formalization purposes. Since our formalization only uses objects that come from a signature as in Definition 2.6, this notion plays the same generating role as that of the standard notion in the literature.

In the formalization, one needs to provide the ingredients from Definition 2.6 to construct an AFS. On pen-and-paper, we denote such an object by $(\mathcal{B}, \mathcal{F}, \mathtt{ar}, \mathcal{R})$, and if the set of base types is clear from the context, we denote an AFS just as $(\mathsf{ar}, \mathcal{R})$. This is a kind of compatibility between the formalization and the pen-and-paper presentation.

Now let us look at a simple example. The base types are lists and natural numbers. The function we look at applies a function F to every element of a list q. Note that this example relies on the fact that we have higher-order types.

▶ **Example 2.7.** We build lists using the constructors $\vdash \mathsf{nil} : \mathsf{nat}$ and $\vdash \mathsf{cons} : \mathsf{nat}{\Rightarrow}\mathsf{list}{\Rightarrow}\mathsf{list}$. The rules for map can be typed using the variable environment $F : \mathsf{nat}{\Rightarrow}\mathsf{nat}, x : \mathsf{nat}$.

$$\mathsf{map}(F, \mathsf{nil}) \to \mathsf{nil} \qquad \mathsf{map}(F, \mathsf{cons}(x, q)) \to \mathsf{cons}(Fx, \mathsf{map}(F, q))$$

3 Higher-Order Interpretation Method

The goal of our tool is to check for termination, and thus theorems that give conditions for strong normalization form the core of the formalization.

Within the higher-order framework, several methods have been developed, and our focus is on the so-called *semantical methods*: to prove an AFS is terminating we need to find a well-founded interpretation domain such that $[\![s]\!] > [\![t]\!]$, whenever $s \to_{\mathcal{R}} t$. This is achieved by orienting each rule in $\mathcal{R}$, that is, $[\![\ell]\!] > [\![r]\!]$ for all rules $\ell \to r$ in $\mathcal{R}$. The idea was first introduced by van de Pol [11] (in the context of HRS) as an extension of the first-order semantic interpretation method. Later, these semantic methods got extended to AFSs by Fuhs and Kop [5] with a special focus on implementation, and it is part of Wanda, a termination tool developed by Kop [8].

In this section, we discuss the main definitions and theorems for such methods. Our notion of well-founded interpretation domain is that of an *extended well-founded set*. These

are sets together with two ordering relations: a well-founded strict $>$ and a quasi-order $\geq$ compatible with it. More precisely, compatible orders are defined as follows.

▶ **Definition 3.1** (Compatible Order)**.** An **extended well-founded set** is a tuple $(X, >, \geq)$ consisting of a set X, a well-founded relation $>$ on X, and a transitive and reflexive relation $\geq$ on X, such that, for all $x, y, z \in X$, the following conditions hold:

- $x > y$ implies $x \geq y$;
- if $x > y$ and $y \geq z$, then we have $x > z$;
- if $x \geq y$ and $y > z$, then we have $x > z$.

To interpret base types, we just need to give an extended well-founded set for each base type. However, we also need to be able to interpret function types. We use *weakly monotonic functions* for that. These are functions that preserve the quasi-ordering $\geq$.

▶ **Definition 3.2.** Given extended well-founded sets X and Y, a **weakly monotonic function** f is a function $f : X \to Y$ such that for all $x, y \in X$, if $x \geq y$, then we have $f(x) \geq f(y)$. If we also have $f(x) > f(y)$ whenever we have $x, y \in X$ such that $x > y$, then we say f a **strongly monotonic function**.

Given two extended well-founded sets X and Y, we can construct another well-founded set whose elements are weakly monotonic functions. Using those, we can define the interpretation $\mathcal{WM}_A$ of a simple type A the same way as Fuhs and Kop [5]. For a full interpretation, we also should interpret terms. The necessary data for that is more complicated, and for the details we refer the reader to [5].

Briefly said, a *weakly monotonic algebra* is needed to interpret types and symbols in $\mathcal{F}$. However, to be able to use it as a reduction ordering additional information is required. Namely, we need a symbol $@^A$, for each type A, satisfying a strictness condition to represent term-application; and symbols in $\mathcal{F}$ has to be interpreted as strongly monotonic functionals. From such an extended algebra, we get maps $[\![\cdot]\!]$ that send terms of type A to elements of $\mathcal{WM}_A$. Using these notions, we can formally prove the main theorem.

▶ **Theorem 3.3** (Compatibility Theorem)**.** *Let an AFS* $(\mathsf{ar}, \mathcal{R})$ *and an extended weakly monotonic algebra for it be given. If for each rewriting rule we have* $[\![\ell]\!] > [\![r]\!]$*, then the AFS is strongly normalizing.*

4 Future Work

We briefly discussed the basics of our formalization of higher-order rewriting in Coq. Up to now, we formalized the basic data structures, namely types, terms, and signatures, and give a formal proof for the compatibility theorem. This theorem is necessary to guarantee the correctness of algorithms that use semantic methods to check for strong normalization.

The next step is to formalize rule removal and actual instances algorithms. We plan to start with higher-order polynomial interpretation. After a full formalization of the tool-chain, we can extract an OCaml program from the Coq implementation, which is guaranteed to satisfy the proven specifications. As a result, we will get a fully verified tool that checks for termination of higher-order programs.

References

1 Bruno Barras et al. The Coq proof assistant reference manual: Version 6.1. Technical report, Inria, 1997. URL: https://hal.inria.fr/inria-00069968.

2 Frédéric Blanqui and Adam Koprowski. CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Math. Struct. Comput. Sci.*, 21:827–859, 2011. doi:10.1017/S0960129511000120.

3 Community. Termination problem database, version 11.0. Directory Higher_Order_Rewriting_Union_Beta/Mixed_HO_10/, 2019. URL: http://termination-portal.org/wiki/TPDB.

4 Evelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Certification of Automated Termination Proofs. In *Proc. FroCoS*, pages 148–162, 2007. doi:10.1007/978-3-540-74621-8_10.

5 Carsten Fuhs and Cynthia Kop. Polynomial Interpretations for Higher-Order Rewriting. In *Proc. RTA*, 2012. doi:10.4230/LIPIcs.RTA.2012.176.

6 Max W. Haslbeck and René Thiemann. An isabelle/hol formalization of aprove's termination method for llvm ir. In *Proc. CPP*, CPP 2021, page 238–249, 2021. doi:10.1145/3437992.3439935.

7 J. Jouannaud and M. Okada. A computation model for executable higher-order algebraic specification languages. In *Proc. LICS*, pages 350–361, 1991. doi:10.1109/LICS.1991.151659.

8 Cynthia Kop. WANDA - a Higher Order Termination Tool (System Description). In *FSCD 2020*, 2020. doi:10.4230/LIPIcs.FSCD.2020.36.

9 Alexander Krauss, Christian Sternagel, René Thiemann, Carsten Fuhs, and Jürgen Giesl. Termination of Isabelle Functions via Termination of Rewriting. In *Proc. ITP*, pages 152–167, 2011. doi:10.1007/978-3-642-22863-6_13.

10 Pierre Letouzey. Extraction in Coq: An Overview. In *Proc. CiE*, pages 359–369, 2008. doi:10.1007/978-3-540-69407-6_39.

11 J.C. van de Pol. *Termination of Higher-order Rewrite Systems*. PhD thesis, University of Utrecht, 1996. URL: https://www.cs.au.dk/~jaco/papers/thesis.pdf.

12 Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq Project. *J Automated Reasoning*, 64:947–999, 2020. doi:10.1007/s10817-019-09540-0.

13 René Thiemann and Christian Sternagel. Certification of Termination Proofs Using CeTA. In *Proc. TPHOLs*, pages 452–468, 2009. doi:10.1007/978-3-642-03359-9_31.

14 Makarius Wenzel, Lawrence C Paulson, and Tobias Nipkow. The Isabelle Framework. In *Proc. TPHOLs*, pages 33–38, 2008. doi:10.1007/978-3-540-71067-7_7.

Observing Loopingness

Étienne Payet
LIM - Université de la Réunion, France

Abstract

In this paper, we consider non-termination in logic programming and in term rewriting and we recall some well-known results for observing it. Then, we instantiate these results to loopingness, a simple form of non-termination. We provide a bunch of examples that seem to indicate that the instantiations are correct as well as partial proofs.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Theory of computation → Rewrite systems; Theory of computation → Program analysis

Keywords and phrases Logic Programming, Term Rewriting Systems, Non-Termination, Loop

1 Introduction

Proving non-termination is an important topic in logic programming and term rewriting. It is also important to determine classes of non-termination and compare them, *e.g.*, in terms of complexity and decidability, for a better understanding of the underlying mechanisms. Loopingness is the simplest form of non-termination and the vast majority of automated techniques for proving non-termination are designed for finding loops. In [10], the more general concept of *inner-loopingness* in term rewriting is introduced and proved undecidable.

Our aim in this paper is to contribute to the understanding of loopingness. We consider some well-known results for observing non-termination in the *unfoldings* and the *chains* and we instantiate them to loopingness. We provide several examples that seem to indicate that the instantiations are correct as well as partial proofs. Observing loopingness (instead of just non-termination) provides clarifications on the non-termination hardness of the program. On the other hand, an observed non-looping non-termination cannot be detected by an automated technique designed for finding loops, hence it is useless to run such a technique for proving this non-termination.

2 Preliminaries

We assume the reader is familiar with the standard definitions of logic programming [1] and term rewriting [3]. We let $\mathbb{N}$ denote the set of non-negative integers. For any set E, we let $\wp(E)$ denote its power set. We let $\xrightarrow{+}$ (resp. $\xrightarrow{*}$) denote the transitive (resp. reflexive and transitive) closure of a binary relation $\rightarrow$. We fix a finite *signature* $\mathcal{F}$ (the *function symbols*) together with an infinite countable set $\mathcal{V}$ of *variables* with $\mathcal{F} \cap \mathcal{V} = \emptyset$. Constant symbols are denoted by $\mathsf{0}, \mathsf{1}, \ldots$, function symbols of positive arity by $\mathsf{f}, \mathsf{g}, \mathsf{s}, \ldots$, variables by $x, y, z, \ldots$ and terms by $l, r, s, t, \ldots$ For any term t, we let $Var(t)$ denote the set of variables occurring in t and $root(t)$ denote the root symbol of t. The set of positions of t is denoted by $Pos(t)$. For any $p \in Pos(t)$, we write $t|_p$ to denote the subterm of t at position p and $t[p \leftarrow s]$ to denote the term obtained from t by replacing $t|_p$ with a term s. A substitution is a finite mapping from variables to terms written as $\{x_1/t_1, \ldots, x_n/t_n\}$. A *(variable) renaming* is a substitution that is a bijection on $\mathcal{V}$. The application of a substitution θ to a syntactic object o (*i.e.*, a construct consisting of terms) is denoted by $o\theta$, and $o\theta$ is called an *instance* of o. When θ is a renaming, $o\theta$ is also called a *variant* of o. The substitution θ is a *unifier* of the syntactic objects o and o' if $o\theta = o'\theta$. We let $mgu(o, o')$ denote the (up to variable renaming)

most general unifier of o and o'. If O is a set of syntactic objects, we write $o \ll O$ to denote that o is a new occurrence of an element of O whose variables are new (not previously met).

2.1 Logic programming

We also fix a finite set of *predicate symbols* disjoint from $\mathcal{F}$ and $\mathcal{V}$ that is used for constructing atoms. Predicate symbols are denoted by $\mathsf{p}, \mathsf{q}, \ldots$, atoms by $H, A, B, \ldots$ and queries (*i.e.*, sequences of atoms) by bold uppercase letters. Consider a non-empty query $\langle A, \mathbf{A}\rangle$ and a clause c. Let $H \leftarrow \mathbf{B}$ be a variant of c variable disjoint with $\langle A, \mathbf{A}\rangle$ and assume that $\theta = mgu(A, H)$. Then $\langle A, \mathbf{A}\rangle \underset{c}{\overset{\theta}{\Rightarrow}} \langle \mathbf{B}, \mathbf{A}\rangle\theta$ is a *derivation step* with $H \leftarrow \mathbf{B}$ as its *input clause*. If the substitution θ or the clause c is irrelevant, we drop a reference to it. For any logic program (LP) P and queries $\mathbf{Q}, \mathbf{Q}'$, we write $\mathbf{Q} \underset{P}{\Rightarrow} \mathbf{Q}'$ if $\mathbf{Q} \underset{c}{\overset{\theta}{\Rightarrow}} \mathbf{Q}'$ holds for some clause $c \in P$ and some substitution θ. A maximal sequence $\mathbf{Q}_0 \underset{P}{\Rightarrow} \mathbf{Q}_1 \underset{P}{\Rightarrow} \cdots$ of derivation steps is called a *derivation of* $P \cup \{\mathbf{Q}_0\}$ if the *standardization apart* condition holds, *i.e.*, each input clause used is variable disjoint from the initial query $\mathbf{Q}_0$ and from the mgu's and input clauses used at earlier steps. We say that a query $\mathbf{Q}$ is *non-terminating* w.r.t. P if there exists an infinite derivation of $P \cup \{\mathbf{Q}\}$. We say that P is *non-terminating* if there exists a query which is non-terminating w.r.t. it.

2.2 Term rewriting

For any terms s and t and any rewrite rule $R = l \to r$, we write $s \underset{R}{\Rightarrow} t$ if there is a substitution θ and a position $p \in Pos(s)$ such that $s|_p = l\theta$ and $t = s[p \leftarrow r\theta]$. Then $s \underset{R}{\Rightarrow} t$ is called a *rewrite step*. For any term rewriting system (TRS) $\mathcal{R}$, we write $s \underset{\mathcal{R}}{\Rightarrow} t$ if $s \underset{R}{\Rightarrow} t$ holds for some $R \in \mathcal{R}$ (then, we also call $s \underset{\mathcal{R}}{\Rightarrow} t$ a rewrite step). A maximal sequence $s_0 \underset{\mathcal{R}}{\Rightarrow} s_1 \underset{\mathcal{R}}{\Rightarrow} \cdots$ of rewrite steps is called a *rewrite of* $\mathcal{R} \cup \{s_0\}$. We say that a term s is *non-terminating* w.r.t. $\mathcal{R}$ if there exists an infinite rewrite of $\mathcal{R} \cup \{s\}$ and we say that $\mathcal{R}$ is *non-terminating* if there exists a term which is non-terminating w.r.t. it.

3 Observing non-termination in logic programming

The *binary unfoldings* [4, 5] transform a LP P into a possibly infinite set of binary clauses. Intuitively, each generated *binary clause* $H \leftarrow B$ (where B is an atom or the empty query true) specifies that, w.r.t. P, a call to H (or any of its instances) necessarily leads to a call to B (or its corresponding instance). A generated clause of the form $H \leftarrow \mathsf{true}$ indicates a success pattern. In the definition below, $\Im$ denotes the domain of binary clauses (viewed modulo renaming) and id denotes the set of all binary clauses of the form $\mathsf{true} \leftarrow \mathsf{true}$ or $\mathsf{p}(x_1, \ldots, x_n) \leftarrow \mathsf{p}(x_1, \ldots, x_n)$, where p is a predicate symbol of arity n and $x_1, \ldots, x_n$ are distinct variables. Given any set X of binary clauses, $T_P^\beta(X)$ is constructed by unfolding prefixes of clause bodies of P, using elements of $X \cup id$, to obtain new binary clauses.

▶ **Definition 1** (Binary unfoldings)**.**

$$T_P^\beta : \wp(\Im) \to \wp(\Im)$$

$$X \mapsto \left\{ (H \leftarrow B)\theta \;\middle|\; \begin{array}{l} c = H \leftarrow B_1, \ldots, B_m \in P,\ i \in \{1, \ldots, m\} \\ \langle H_j \leftarrow \mathsf{true}\rangle_{j=1}^{i-1} \ll X \\ H_i \leftarrow B \ll X \cup id,\ i < m \Rightarrow B \neq \mathsf{true} \\ \theta = mgu(\langle B_1, \ldots, B_i\rangle, \langle H_1, \ldots, H_i\rangle) \end{array} \right\}$$

and $unf(P) = \bigcup_{n\in\mathbb{N}}(T_P^\beta)^n(\emptyset)$, *where* $(T_P^\beta)^0(\emptyset) = \emptyset$.

▶ **Example 2.** Consider the logic program P that consists of the clauses

$$c_1 = \mathsf{p}(x,y) \leftarrow \mathsf{q}(x), \mathsf{p}(y,x) \qquad c_2 = \mathsf{q}(\mathsf{0}) \leftarrow \mathsf{true}$$

Unfolding c_2 using $\mathsf{true} \leftarrow \mathsf{true} \in id$, one gets $c_2' = \mathsf{q}(\mathsf{0}) \leftarrow \mathsf{true} \in T_P^\beta(\emptyset)$. Then, unfolding c_1 using c_2', $\mathsf{p}(x',y') \leftarrow \mathsf{p}(x',y') \in id$ and $i = 2$, one gets $c_3 = \mathsf{p}(\mathsf{0},y) \leftarrow \mathsf{p}(y,\mathsf{0}) \in (T_P^\beta)^2(\emptyset)$. Finally, unfolding c_1 using c_2', c_3 and $i = 2$, one gets $c_4 = \mathsf{p}(\mathsf{0},\mathsf{0}) \leftarrow \mathsf{p}(\mathsf{0},\mathsf{0}) \in (T_P^\beta)^3(\emptyset)$.

It is proved in [4] that the binary unfoldings of a LP exhibit its termination properties:

▶ **Theorem 3** (Observing non-termination in the unfoldings). *Let P be a LP and $\mathbf{Q}$ be a query. Then, $\mathbf{Q}$ is non-terminating w.r.t. P iff $\mathbf{Q}$ is non-terminating w.r.t. $unf(P)$.*

For instance, in Ex. 2, we have $c_4 = \mathsf{p}(\mathsf{0},\mathsf{0}) \leftarrow \mathsf{p}(\mathsf{0},\mathsf{0}) \in unf(P)$, so the query $\mathsf{p}(\mathsf{0},\mathsf{0})$ is non-terminating w.r.t. $unf(P)$. Hence, by Thm. 3, $\mathsf{p}(\mathsf{0},\mathsf{0})$ is non-terminating w.r.t. P.

The proof of Thm. 3 relies on the following definition and theorem.

▶ **Definition 4** (Calls-to relation $\rightsquigarrow$). *Let P be a LP. For any atoms A and B, we say that B is a* call *in a derivation of $P \cup \{A\}$, denoted $A \underset{P}{\rightsquigarrow} B$, if $A \overset{+}{\underset{P}{\Rightarrow}} \langle B, \dots\rangle$; we also write $A \underset{L}{\rightsquigarrow} B$ to emphasize that L is the sequence of clauses of P used in a derivation from A to $\langle B, \dots\rangle$. A* P-chain *is a (possibly infinite) sequence of the form $A_0 \underset{P}{\rightsquigarrow} A_1 \underset{P}{\rightsquigarrow} A_2 \underset{P}{\rightsquigarrow} \cdots$*

▶ **Theorem 5** (Observing non-termination in the chains). *A LP P is non-terminating iff there exists an infinite P-chain.*

For instance, in Ex. 2, we have the infinite P-chain $\mathsf{p}(\mathsf{0},\mathsf{0}) \underset{P}{\rightsquigarrow} \mathsf{p}(\mathsf{0},\mathsf{0}) \underset{P}{\rightsquigarrow} \cdots$

4 Observing non-termination in term rewriting

We consider the unfolding technique used in [8]. It is defined as a function over the domain $\Re$ of rewrite rules (viewed modulo renaming). It is based on forward and backward narrowing and also performs unfolding on variable positions (contrary to what is usually done in the literature). Note that in general, the unfoldings of a TRS are not finitely computable.

▶ **Definition 6** (Unfoldings).

$$U_\mathcal{R} : \wp(\Re) \to \wp(\Re)$$

$$X \mapsto \underbrace{\left\{ (l \to r[p \leftarrow r'])\theta \;\middle|\; \begin{array}{l} l \to r \in X \\ p \in Pos(r) \\ l' \to r' \ll \mathcal{R} \\ \theta = mgu(r|_p, l') \end{array} \right\}}_{\textit{forward unfoldings}} \cup \underbrace{\left\{ (l[p \leftarrow l'] \to r)\theta \;\middle|\; \begin{array}{l} l \to r \in X \\ p \in Pos(l) \\ l' \to r' \ll \mathcal{R} \\ \theta = mgu(l|_p, r') \end{array} \right\}}_{\textit{backward unfoldings}}$$

and $unf(\mathcal{R}) = \bigcup_{n\in\mathbb{N}}(U_\mathcal{R})^n(\mathcal{R})$, *where* $(U_\mathcal{R})^0(\mathcal{R}) = \mathcal{R}$.

▶ **Example 7.** Consider the TRS $\mathcal{R}$ introduced by Toyama [9] that consists of the rules

$$R_1 = \mathsf{f}(\mathsf{0},\mathsf{1},x) \to \mathsf{f}(x,x,x) \quad R_2 = \mathsf{g}(x,y) \to x \quad R_3 = \mathsf{g}(x,y) \to y$$

We have $R_1 \in (U_\mathcal{R})^0(\mathcal{R})$. Unfolding R_1 backwards using R_2 and $p = 1$, one gets $R_4 = \mathsf{f}(\mathsf{g}(\mathsf{0},y'),\mathsf{1},x) \to \mathsf{f}(x,x,x) \in U_\mathcal{R}(\mathcal{R})$. Then, unfolding R_4 backwards using R_3 and $p = 2$, one gets $R_5 = \mathsf{f}(\mathsf{g}(\mathsf{0},y'),\mathsf{g}(x'',\mathsf{1}),x) \to \mathsf{f}(x,x,x) \in (U_\mathcal{R})^2(\mathcal{R})$.

By [6], for all $s \to t \in \mathit{unf}(\mathcal{R})$ we have $s \overset{+}{\underset{\mathcal{R}}{\Rightarrow}} t$. So, as $\mathcal{R} \subseteq \mathit{unf}(\mathcal{R})$ also holds, the unfoldings of a TRS exhibit its termination properties:

▶ **Theorem 8** (Observing non-termination in the unfoldings). *Let $\mathcal{R}$ be a TRS and s be a term. Then, s is non-terminating w.r.t. $\mathcal{R}$ iff s is non-terminating w.r.t. $\mathit{unf}(\mathcal{R})$.*

In Ex. 7 above, we have $R_5 = \mathsf{f}(\mathsf{g}(0, y'), \mathsf{g}(x'', 1), x) \to \mathsf{f}(x, x, x) \in \mathit{unf}(\mathcal{R})$, hence the term $s = \mathsf{f}(\mathsf{g}(0,1), \mathsf{g}(0,1), \mathsf{g}(0,1))$ is non-terminating w.r.t. $\mathit{unf}(\mathcal{R})$ (we have $s \underset{R_5}{\Rightarrow} s \underset{R_5}{\Rightarrow} \cdots$). Consequently, by Thm. 8, s is non-terminating w.r.t. $\mathcal{R}$.

We refer to [2] for details on dependency pairs. The *defined symbols* of a TRS $\mathcal{R}$ are $\mathcal{D}_\mathcal{R} = \{\mathit{root}(l) \mid l \to r \in \mathcal{R}\}$. For every $\mathsf{f} \in \mathcal{F}$ we let $\mathsf{f}^\#$ be a fresh *tuple symbol* with the same arity as f. If $t = \mathsf{f}(t_1, \ldots, t_m)$ is a term, we let $t^\#$ denote the construct $\mathsf{f}^\#(t_1, \ldots, t_m)$. The set of *dependency pairs* of $\mathcal{R}$ is $DP(\mathcal{R}) = \{l^\# \to t^\# \mid l \to r \in \mathcal{R},\ t \text{ is a subterm of } r,\ \mathit{root}(t) \in \mathcal{D}_\mathcal{R}\}$ (viewed modulo renaming). A (possibly infinite) sequence $\mathfrak{C} = \langle s_1^\# \to t_1^\#, s_2^\# \to t_2^\#, \ldots \rangle$ of dependency pairs of $\mathcal{R}$ is an *$\mathcal{R}$-chain* if there exist substitutions σ_i such that $t_i^\#\sigma_i \overset{*}{\underset{\mathcal{R}}{\Rightarrow}} s_{i+1}^\#\sigma_{i+1}$ holds for every two consecutive pairs $s_i^\# \to t_i^\#$ and $s_{i+1}^\# \to t_{i+1}^\#$ in the sequence. We may also write $\mathfrak{C}$ as $\langle (s_1^\# \to t_1^\#, \sigma_1), (s_2^\# \to t_2^\#, \sigma_2), \ldots \rangle$ to emphasize that $\sigma_1, \sigma_2, \ldots$ are substitutions associated with every two consecutive pairs. It is proved in [2] that the presence of an infinite $\mathcal{R}$-chain is a sufficient and necessary criterion for non-termination:

▶ **Theorem 9** (Observing non-termination in the chains). *A TRS $\mathcal{R}$ is non-terminating* iff *there exists an infinite $\mathcal{R}$-chain.*

For instance, in Ex. 7, $\langle \mathsf{f}^\#(0,1,x) \to \mathsf{f}^\#(x,x,x), \mathsf{f}^\#(0,1,x) \to \mathsf{f}^\#(x,x,x), \ldots \rangle$ is an infinite $\mathcal{R}$-chain because, for $\sigma = \{x/\,\mathsf{g}(0,1)\}$, we have $\mathsf{f}^\#(x,x,x)\sigma \overset{*}{\underset{\mathcal{R}}{\Rightarrow}} \mathsf{f}^\#(0,1,x)\sigma$.

5 Observing loopingness

The definitions presented below hold both in logic programming and in term rewriting, so we introduce a generic terminology. By a *program* (denoted by Π, $\Pi' \ldots$) we mean a LP or a TRS, by a *rule* (denoted by π, $\pi' \ldots$) we mean a clause or a rewrite rule, by a *goal* (denoted by α, $\alpha' \ldots$) we mean a query or a term, by a *computation* we mean a derivation or a rewrite.

Let $L = \langle \pi_1, \ldots, \pi_n \rangle$ be a finite non-empty sequence of rules. For any goals α, α' we write $\alpha \underset{L}{\hookrightarrow} \alpha'$ when $\alpha \underset{\pi_1}{\Rightarrow} \cdots \underset{\pi_n}{\Rightarrow} \alpha'$.

▶ **Definition 10** (Looping). *Let Π be a program, L be a finite non-empty sequence of rules of Π and α be a goal. We say that a computation of $\Pi \cup \{\alpha\}$ is L-*looping *if it is infinite and has the form $\alpha \underset{L}{\hookrightarrow} \alpha_1 \underset{L}{\hookrightarrow} \alpha_2 \underset{L}{\hookrightarrow} \cdots$. We may drop the reference to L if it is not relevant, and simply say that the computation is* looping. *We say that α is* looping *w.r.t. Π if there exists a looping computation of $\Pi \cup \{\alpha\}$. We say that Π is* looping *if there exists a goal which is looping w.r.t. it.*

▶ **Example 11.** Consider the LP P which consists of the clauses $c_1 = \mathsf{p}_1 \leftarrow \mathsf{p}_2$, $c_2 = \mathsf{p}_2 \leftarrow \mathsf{p}_3$, $c_3 = \mathsf{p}_3 \leftarrow \mathsf{p}_1$ and $c_4 = \mathsf{p}_3 \leftarrow \mathsf{p}_4$. Then, p_1 is looping w.r.t. P as we have the infinite derivation $\mathsf{p}_1 \underset{c_1}{\Rightarrow} \mathsf{p}_2 \underset{c_2}{\Rightarrow} \mathsf{p}_3 \underset{c_3}{\Rightarrow} \mathsf{p}_1 \underset{c_1}{\Rightarrow} \mathsf{p}_2 \underset{c_2}{\Rightarrow} \mathsf{p}_3 \underset{c_3}{\Rightarrow} \mathsf{p}_1 \underset{c_1}{\Rightarrow} \cdots$ *i.e.*, for $L = \langle c_1, c_2, c_3 \rangle$, $\mathsf{p}_1 \underset{L}{\hookrightarrow} \mathsf{p}_1 \underset{L}{\hookrightarrow} \mathsf{p}_1 \underset{L}{\hookrightarrow} \cdots$

We extend the concept of loopingness to chains.

▶ **Definition 12** (Looping chain). *Let P be a LP and $\mathcal{R}$ be a TRS.*

- *We say that a P-chain is* looping *if it is infinite and has the form $A_0 \underset{L}{\rightsquigarrow} A_1 \underset{L}{\rightsquigarrow} \cdots$ where L is a finite, non-empty, sequence of clauses of P.*
- *We say that an $\mathcal{R}$-chain is* looping *if it is infinite and has the form $\langle L, L, \dots \rangle$, where L is a finite, non-empty, sequence of elements of $DP(\mathcal{R}) \times$ Substitutions.*

▶ **Example 13** (Ex. 7 continued)**.** Let $p = \big(\mathsf{f}^{\#}(0,1,x) \to \mathsf{f}^{\#}(x,x,x), \{x/\,\mathsf{g}(0,1)\}\big)$. Then, $\langle p, p, \dots \rangle$ is a looping $\mathcal{R}$-chain.

Note that there exist infinite computations which are not looping, *i.e.*, do not correspond to the infinite repetition of the same sequence of rules.

▶ **Example 14.** Let P be the LP which consists of the clauses $c_1 = \mathsf{p}(0,y) \leftarrow \mathsf{p}(\mathsf{s}(y),\mathsf{s}(y))$ and $c_2 = \mathsf{p}(\mathsf{s}(x),y) \leftarrow \mathsf{p}(x,y)$. We have the following infinite derivation of $P \cup \{\mathsf{p}(0,0)\}$:

$$\mathsf{p}(0,0) \underset{c_1}{\Rightarrow} \mathsf{p}(\mathsf{s}(0),\mathsf{s}(0)) \underset{c_2}{\Rightarrow} \mathsf{p}(0,\mathsf{s}(0)) \underset{c_1}{\Rightarrow} \mathsf{p}(\mathsf{s}^2(0),\mathsf{s}^2(0)) \overset{2}{\underset{c_2}{\Rightarrow}} \mathsf{p}(0,\mathsf{s}^2(0)) \underset{c_1}{\Rightarrow} \cdots$$

It is not looping as it follows the path $\langle c_1, c_2, c_1, c_2, c_2, c_1, \dots \rangle$. We also have the infinite, non-looping, P-chain:

$$\mathsf{p}(0,0) \underset{c_1}{\rightsquigarrow} \mathsf{p}(\mathsf{s}(0),\mathsf{s}(0)) \underset{c_2}{\rightsquigarrow} \mathsf{p}(0,\mathsf{s}(0)) \underset{c_1}{\rightsquigarrow} \mathsf{p}(\mathsf{s}^2(0),\mathsf{s}^2(0)) \underset{\langle c_2,c_2\rangle}{\rightsquigarrow} \mathsf{p}(0,\mathsf{s}^2(0)) \underset{c_1}{\rightsquigarrow} \cdots$$

▶ **Example 15.** Let $\mathcal{R}$ be the TRS which consists of the rules $R_1 = \mathsf{f}(0,y) \to \mathsf{f}(\mathsf{s}(y),\mathsf{s}(y))$ and $R_2 = \mathsf{f}(\mathsf{s}(x),y) \to \mathsf{f}(x,y)$. We have the following infinite rewrite of $\mathcal{R} \cup \{\mathsf{f}(0,0)\}$:

$$\mathsf{f}(0,0) \underset{R_1}{\Rightarrow} \mathsf{f}(\mathsf{s}(0),\mathsf{s}(0)) \underset{R_2}{\Rightarrow} \mathsf{f}(0,\mathsf{s}(0)) \underset{R_1}{\Rightarrow} \mathsf{f}(\mathsf{s}^2(0),\mathsf{s}^2(0)) \overset{2}{\underset{R_2}{\Rightarrow}} \mathsf{f}(0,\mathsf{s}^2(0)) \underset{R_1}{\Rightarrow} \cdots$$

It is not looping as it follows the path $\langle R_1, R_2, R_1, R_2, R_2, R_1, \dots \rangle$. We also have the infinite, non-looping, $\mathcal{R}$-chain $\langle (R_1^{\#}, \sigma_1), (R_2^{\#}, \theta_1), (R_1^{\#}, \sigma_2), (R_2^{\#}, \theta_2), \dots \rangle$ where $R_1^{\#} = s_1^{\#} \to t_1^{\#} = \mathsf{f}^{\#}(0,y) \to \mathsf{f}^{\#}(\mathsf{s}(y),\mathsf{s}(y))$, $R_2^{\#} = s_2^{\#} \to t_2^{\#} = \mathsf{f}^{\#}(\mathsf{s}(x),y) \to \mathsf{f}^{\#}(x,y)$ and, for all $i > 0$, $\sigma_i = \{y/\,\mathsf{s}^{i-1}(0)\}$ and $\theta_i = \{x/0, y/\,\mathsf{s}^i(0)\}$. Indeed, we have $t_1^{\#}\sigma_1 \overset{*}{\underset{\mathcal{R}}{\Rightarrow}} s_2^{\#}\theta_1$, $t_2^{\#}\theta_1 \overset{*}{\underset{\mathcal{R}}{\Rightarrow}} s_1^{\#}\sigma_2$, ...

All the examples given above seem to indicate that the *Observing non-termination* results of Sect. 3 and Sect. 4 can be instantiated to loopingness.

▶ **Lemma 16** (Observing loopingness in the chains)**.** *If, for a program* Π*, there exists a looping* Π*-chain then* Π *is looping.*

Proof. For LPs, the result immediately follows from Def. 4 and Def. 12. For TRSs, it is proved in [2] that any infinite Π-chain $\mathfrak{C} = \langle (s_1^{\#} \to t_1^{\#}, \sigma_1), (s_2^{\#} \to t_2^{\#}, \sigma_2), \dots \rangle$ corresponds to an infinite rewrite $\mathfrak{C}' = (s_1\sigma_1 \underset{R_1}{\Rightarrow} C_1[t_1]\sigma_1 \overset{*}{\underset{\Pi}{\Rightarrow}} C_1[s_2]\sigma_2 \underset{R_2}{\Rightarrow} C_1[C_2[t_2]]\sigma_2 \overset{*}{\underset{\Pi}{\Rightarrow}} \cdots)$ where $R_1 = s_1 \to C_1[t_1]$, $R_2 = s_2 \to C_2[t_2]$, ... are rewrite rules of Π and the rewrites in $\overset{*}{\underset{\Pi}{\Rightarrow}}$ do not occur in the C_i's (*i.e.*, they are those of $t_i^{\#}\sigma_i \overset{*}{\underset{\Pi}{\Rightarrow}} s_{i+1}^{\#}\sigma_{i+1}$). Hence, if $\mathfrak{C}$ is looping, so is $\mathfrak{C}'$. ◀

▶ **Conjecture 17** (Observing loopingness in the chains)**.** *If a program* Π *is looping then there exists a looping* Π*-chain.*

Proof sketch for a LP P**.** Let $\mathbf{Q}_0 \underset{L}{\hookrightarrow} \mathbf{Q}_1 \underset{L}{\hookrightarrow} \cdots$ be a looping derivation of $P \cup \{\mathbf{Q}_0\}$. Then, in each step $\mathbf{Q}_i \underset{L}{\hookrightarrow} \mathbf{Q}_{i+1}$ there is a query $\langle A, \dots \rangle$ that has an infinite derivation. For all $i \in \mathbb{N}$, let $\langle A_i, \dots \rangle$ be the leftmost such query in $\mathbf{Q}_i \underset{L}{\hookrightarrow} \mathbf{Q}_{i+1}$. Then, for all $i \in \mathbb{N}$ we have $A_i \underset{P}{\rightsquigarrow} A_{i+1}$. Let L' be the sequence of clauses used in $A_0 \overset{+}{\underset{P}{\Rightarrow}} \langle A_1, \dots \rangle$. We prove by induction on i that, for all $i \in \mathbb{N}$, L' is used in $A_i \overset{+}{\underset{P}{\Rightarrow}} \langle A_{i+1}, \dots \rangle$ and hence that $A_i \underset{L'}{\rightsquigarrow} A_{i+1}$. ◀

▶ **Lemma 18** (Observing loopingness in the unfoldings). *A term is looping w.r.t. a TRS $\mathcal{R}$ iff it is looping w.r.t. $unf(\mathcal{R})$.*

Proof. ($\Rightarrow$) As $\mathcal{R} \subseteq unf(\mathcal{R})$, any rewrite with $\mathcal{R}$ is also a rewrite with $unf(\mathcal{R})$. ($\Leftarrow$) For all $s \to t \in unf(\mathcal{R})$ we have $s \overset{+}{\underset{\mathcal{R}}{\Rightarrow}} t$ [6], so replacing, in a looping rewrite with $unf(\mathcal{R})$, each step by the corresponding finite sequence of steps in $\mathcal{R}$, one gets a looping rewrite with $\mathcal{R}$. ◀

▶ **Conjecture 19** (Observing loopingness in the binary unfoldings). *A query is looping w.r.t. a LP P iff it is looping w.r.t. $unf(P)$.*

Proof sketch. Use Lem. 16 + Conj. 17 and the fact that $\underset{P}{\rightsquigarrow} = \underset{unf(P)}{\rightsquigarrow}$ [4]. ◀

▶ **Example 20.** In Ex. 2, we have the $\langle c_4 \rangle$-looping derivation $\mathsf{p}(0,0) \underset{c_4}{\Rightarrow} \mathsf{p}(0,0) \underset{c_4}{\Rightarrow} \cdots$ of $unf(P) \cup \{\mathsf{p}(0,0)\}$ and the $\langle c_1, c_2 \rangle$-looping derivation $\mathsf{p}(0,0) \underset{c_1}{\Rightarrow} \langle \mathsf{q}(0), \mathsf{p}(0,0) \rangle \underset{c_2}{\Rightarrow} \mathsf{p}(0,0) \underset{c_1}{\Rightarrow} \cdots$ of $P \cup \{\mathsf{p}(0,0)\}$. Note that $c_1 \notin unf(P)$ (c_1 is not binary) hence this derivation of $P \cup \{\mathsf{p}(0,0)\}$ is not a derivation of $unf(P) \cup \{\mathsf{p}(0,0)\}$. In Ex. 7, for $s = \mathsf{f}(\mathsf{g}(0,1), \mathsf{g}(0,1), \mathsf{g}(0,1))$, we have the $\langle R_5 \rangle$-looping rewrite $s \underset{R_5}{\Rightarrow} s \underset{R_5}{\Rightarrow} \cdots$ of $unf(\mathcal{R}) \cup \{s\}$ and the $\langle R_2, R_3, R_1 \rangle$-looping rewrite $s \underset{R_2}{\Rightarrow} \mathsf{f}(0, \mathsf{g}(0,1), \mathsf{g}(0,1)) \underset{R_3}{\Rightarrow} \mathsf{f}(0,1,\mathsf{g}(0,1)) \underset{R_1}{\Rightarrow} s \underset{R_2}{\Rightarrow} \cdots$ of $\mathcal{R} \cup \{s\}$. As $\mathcal{R} \subseteq unf(\mathcal{R})$, this rewrite of $\mathcal{R} \cup \{s\}$ is also a rewrite of $unf(\mathcal{R}) \cup \{s\}$.

6 Acknowledgement and future work

We thank the anonymous referees for their valuable comments and constructive criticisms.

Besides finishing the proofs (Conj. 19 and Conj. 17), we plan to extend the results to dependency pairs in logic programming [7] and to inner-loopingness [10]. We also plan to unify more concepts from termination analysis of LPs and TRSs.

References

1 K. R. Apt. *From logic programming to Prolog*. Prentice Hall International series in computer science. Prentice Hall, 1997.

2 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236:133–178, 2000. doi:10.1016/S0304-3975(99)00207-8.

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

4 M. Codish and C. Taboch. A semantic basis for the termination analysis of logic programs. *Journal of Logic Programming*, 41(1):103–123, 1999. doi:10.1016/S0743-1066(99)00006-0.

5 M. Gabbrielli and R. Giacobazzi. Goal independency and call patterns in the analysis of logic programs. In H. Berghel, T. Hlengl, and J. E. Urban, editors, *Proc. of SAC'94*, pages 394–399. ACM Press, 1994. doi:10.1145/326619.326789.

6 J. V. Guttag, D. Kapur, and D. R. Musser. On proving uniform termination and restricted termination of rewriting systems. *SIAM Journal of Computing*, 12(1):189–214, 1983. doi:10.1137/0212012.

7 M. T. Nguyen, J. Giesl, P. Schneider-Kamp, and D. De Schreye. Termination analysis of logic programs based on dependency graphs. In A. King, editor, *Proc. of LOPSTR'07*, volume 4915 of *LNCS*, pages 8–22. Springer, 2007. doi:10.1007/978-3-540-78769-3_2.

8 É. Payet. Loop detection in term rewriting using the eliminating unfoldings. *Theoretical Computer Science*, 403(2-3):307–327, 2008. doi:10.1016/j.tcs.2008.05.013.

9 Y. Toyama. Counterexamples to the termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25(3):141–143, 1987. doi:10.1016/0020-0190(87)90122-0.

10 Y. Wang and M. Sakai. On non-looping term rewriting. In A. Geser and H. Søndergaard, editors, *Proc. of WST'06*, pages 17–21, 2006.