Joint Proceedings of HOR 2019 and IWC 2019
(with system descriptions from CoCo 2019)

Contents

Preface

This report contains the joint proceedings of the 10th Workshop on Higher-Order Rewriting (HOR) and 7th International Workshop on Confluence (IWC), held in Dortmund, Germany on June 28th, 2019. In addition, the proceedings include the system descriptions of the 8th Confluence Competition (CoCo 2019). The workshops were part of the International Conference on Formal Structures for Computation and Deduction (FSCD 2019).

HOR is a forum to present work concerning all aspects of higher-order rewriting. The aim is to provide an informal and friendly setting to discuss recent work and work in progress concerning higher-order rewriting, broadly construed. This includes various topics of interest that range from foundations (pattern matching, unification, strategies, narrowing, termination, syntactic properties, type theory), frameworks (term rewriting, conditional rewriting, graph rewriting, net rewriting), semantics (operational semantics, denotational semantics, separability, higher-order abstract syntax) to implementation (graphs, nets, abstract machines, explicit substitution, rewriting tools, compilation techniques) and application (proof checking, theorem proving, generic programming, declarative programming, program transformation, certification).

Confluence provides a general notion of determinism and has been conceived as one of the central properties of rewriting. Confluence relates to the many topics of rewriting (completion, termination, commutation, coherence, etc.) and has been investigated in many formalisms of rewriting such as first-order rewriting, lambda-calculi, higher-order rewriting, higher-dimensional rewriting, constrained rewriting, conditional rewriting, etc. Recently, there is a renewed interest in confluence research, resulting in new techniques, tool support, certification, as well as applications. IWC promotes and stimulates research and collaboration on confluence and related properties.

The joint program contains 5 contributed talks as well as invited talks by Martin Avanzini, Fransico Duran, Jörg Endrullis, Cynthia Kop, and Giulio Manzonetto. In addition, the program contains the system descriptions from the 8th Confluence Competition (CoCo 2019) held in conjunction with the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019).

Many people contributed to the preparations of HOR and IWC. Hard work by the program commitees, steering committees, and subreviewers made an exciting program of contributed and invited talks possible. In addition, we are greatful to the organizing committee and workshop chairs of FSCD for hosting the workshops in Dortmund.

June 14, 2019 Brasilia/Novi Sad/Copenhagen
Mauricio Ayala-Rincón
Silvia Ghilezan
Jakob Grue Simonsen

HOR Program Committee

Silvia Ghilezan (chair)	University of Novi Sad, Serbia
Stefano Guerrini	Université Paris 13, France
Masahito Hasegawa	Kyoto University, Japan
Cynthia Kop	Radboud University, The Netherlands
Pierre Lescanne	École Normale Superieure de Lyon, France
Julian Nagele	Queen Mary University of London, United Kingdom
Vincent van Oostrom	University of Innsbruck, Austria

HOR Steering committee

Delia Kesner	Université Paris 7, France
Femke van Raamsdonk	Free University of Amsterdam, The Netherlands

IWC Program Committee

Sandra Alves	University of Porto, Protugal
Mauricio Ayala-Rincón (co-chair)	University of Brasilia, Brazil
Cyrille Chenavier	INRIA Lille, France
Alejandro Díaz-Caro	University of Quilmes and ICC/UBA-CONICET, Argentina
Jörg Endrullis	Free University of Amsterdam, The Netherlands
Jakob Grue Simonsen (co-chair)	University of Copenhagen, Denmark
Raúl Guttiérez	Technical University of València, Spain
Camilo Rocha	Pontifical Xavierian University, Colombia
Masahiko Sakai	University of Nagoya, Japan
Sarah Winkler	University of Innsbruck, Austria

IWC subreviewer: Guido Martinez.

IWC Steering Committee

Takahito Aoto	Niigata University, Japan
Nao Hirokawa	Japan Advanced Institute of Science and Technology, Japan

HOR 2019

Higher-Order Complexity Analysis With First-Order Tools

Martin Avanzini[1], Ugo Dal Lago[12], and Georg Moser[3] *

[1] INRIA Sophia Antipolis, France
[2] Università di Bologna, Italy
[3] University of Innsbruck, Austria

1 Introduction

Automatically checking programs for correctness has attracted the attention of the computer science research community since the birth of the discipline. Properties of interest are not necessarily functional, however, and among the non-functional ones, noticeable cases are bounds on the amount of resources (like time, memory and power) programs need when executed.

Deriving upper bounds on the resource consumption of programs is indeed of paramount importance in many cases, but becomes undecidable as soon as the underlying programming language is non-trivial. If the units of measurement become concrete and close to the physical ones, the problem gets even more complicated, given the many transformation and optimisation layers programs are applied to before being executed. A typical example is the one of WCET techniques adopted in real-time systems [22], which do not only need to deal with how many machine instructions a program corresponds to, but also with how much time each instruction costs when executed by possibly complex architectures (including caches, pipelining, etc.), a task which is becoming even harder with the current trend towards multicore architectures.

As an alternative, one can analyse the *abstract* complexity of programs. For instance, one can take the number of instructions executed by the program or the number of evaluation steps to normal form, as a measure of its execution time. This is a less informative metric, which however becomes accurate if the actual time complexity *of each instruction* is kept low. One advantage of this analysis is the independence from the specific hardware platform executing the program at hand: the latter only needs to be analysed once. This is indeed a path which many have followed in the programming language community. A variety of verification techniques have been employed in this context, like abstract interpretations, model checking, type systems, program logics, or interactive theorem provers; see [1, 13, 21] for some pointers. If we restrict our attention to higher-order functional programs, however, the literature becomes much sparser.

The rewriting-community has recently developed several tools for the automated time complexity analysis of *term rewrite system*, a formal model of computation that is at the heart of functional programming. Examples are AProVE [7], and TCT [3]. These *first-order provers* (FOPs for short) combine many different techniques, and after some years of development, start being able to treat non-trivial programs, as demonstrated by the result of the annual termination competition.[1] Such tools are potentially very interesting also for the complexity analysis of *higher-order functional programs*, since well-known transformation techniques such as *defunctionalisation* [20] are available, which turn higher-order functional programs into equivalent first-order ones. This has been done in the realm of termination [18, 8], but appears to be infeasible in the context of complexity analysis. Conclusively this program transformation approach has been reflected critically in the literature, cf. [13].

*This work was partially supported by FWF project number J3563, FWF project number P25781 and by French ANR project Elica ANR-14-CE25-0005.

[1] http://termination-portal.org/wiki/Termination_Competition.

A natural question, then, is whether time complexity analysis of higher-order programs can indeed be performed by going through first-order tools. Is it possible to evaluate the unitary cost of functional programs by translating them into first-order programs, analysing them by existing first-order tools, and thus obtaining meaningful and informative results? Is, e.g., plain defunctionalisation enough? As it turns out, defunctionalisation is indeed not enough, primarily, because the resulting system lacks sufficient structure to be analysed automatically by current FOPs. However, the situation can be rectified by post-processing the defunctionalised program.

In the following, we give an informal account on how FOPs can be leveraged to reason about the runtime complexity of higher-order programs. Also, we briefly report on a tool, duped HoCA, implementing the discussed procedures.[2] More details and formal definitions can be found in the corresponding paper [2].

2 On Defunctionalisation: Ruling the Chaos

The main idea behind defunctionalisation is conceptually simple: function-abstractions are represented as first-order values; calls to abstractions are replaced by calls to a globally defined *apply-function*. Consider for instance the following OCaml-program:

```
let comp f g = fun z → f (g z);;
let rec walk xs =
  match xs with
    [] → (fun z → z)
  | x :: ys → comp (walk ys) (fun z → x::z);;
let rev l = walk l [];;
let main l = rev l;;
```

Run on a list of n elements, `walk` first constructs a function which reverses its first argument and appends it to the second argument. This function, which can be easily defined by recursion, is fed in `rev` with the empty list. The function `main` only serves the purpose of indicating the complexity of *which* function we are interested at.

Defunctionalisation can be understood already at this level. We first define a datatype for representing the three abstractions occurring in the program:

```
type 'a cl = C1 of 'a cl * 'a cl   (* fun z → f (g z) *)
           | C2                    (* fun z → z *)
           | C3 of 'a              (* fun z → x::z *)
```

More precisely, an expression of type `'a cl` represents a function *closure*, whose arguments are used to store assignments to free variables. An infix operator (`@`), modelling application, can then be defined that evaluates these closures.[3] Overall, we arrive at a first-order version of the original higher-order function:

```
let rec (@) c z =
  match c with
    C1(f,g) → f @ (g @ z)
  | C2 → z
  | C3(x) → x::z;;
let comp f g = C1(f,g);;
```

```
let rec walk xs =
  match xs with
    [] → C2
  | x::ys → comp (walk ys) C3(x);;
let rev l = walk l @ [];;
let main l = rev l;;
```

[2] Our tool HoCA is open source and available under http://cbr.uibk.ac.at/tools/hoca/.
[3] The definition of (`@`) is rejected by the OCaml type-checker, which however, is not an issue in our context.

Observe that now the recursive function `walk` constructs an explicit representation of the closure computed by its original definition. The function (`@`) carries out the remaining evaluation. This program can now already be understood as a first-order rewrite system.

Of course, a systematic construction of the defunctionalised program requires some care. For instance, one has to deal with closures that originate from partial function applications. Still, the construction is quite easy to mechanize. On our running example, this program transformation results in the rewrite system $\mathcal{A}_{\mathsf{rev}}$, which looks as follows:

$$
\begin{array}{llll}
(1) & \mathtt{main}(l) \to \mathtt{Rev} \mathbin{@} l & (2) & \mathtt{Rev} \mathbin{@} l \to \mathtt{Fix}_{\mathtt{w}} \mathbin{@} l \mathbin{@} [] \\
(3) & \mathtt{Fix}_{\mathtt{w}} \mathbin{@} xs \to \mathtt{Walk} \mathbin{@} xs & (4) & \mathtt{Walk} \mathbin{@} xs \to \mathtt{match}_{\mathtt{w}}(xs) \\
(5) & \mathtt{match}_{\mathtt{w}}([]) \to \mathtt{C2} & (6) & \mathtt{match}_{\mathtt{w}}(x{::}ys) \to \mathtt{Comp} \mathbin{@} (\mathtt{Fix}_{\mathtt{w}} \mathbin{@} ys) \mathbin{@} \mathtt{C3}(x) \\
(7) & \mathtt{Comp} \mathbin{@} f \to \mathtt{Comp}_1(f) & (8) & \mathtt{Comp}_1(f) \mathbin{@} g \to \mathtt{C1}(f,g) \\
(9) & \mathtt{C1}(f,g) \mathbin{@} z \to f \mathbin{@} (g \mathbin{@} z) & (10) & \mathtt{C2} \mathbin{@} z \to z \\
(11) & \mathtt{C3}(x) \mathbin{@} z \to x{::}z\ . & &
\end{array}
$$

Despite its conceptual simplicity, current FOPs are unable to effectively analyse *applicative* rewrite systems, such as the one above. The reason this happens lies in the way FOPs work, which itself reflects the state of the art on formal methods for complexity analysis of first-order rewrite systems. In order to achieve composability of the analysis, the given system is typically split into smaller parts (see for example [4]), and each of them is analysed separately. Furthermore, contextualisation (aka *path analysis* [11]) and a suitable form of flow graph analysis (or *dependency pair analysis* [10, 16]) is performed. However, at the end of the day, syntactic and semantic basic techniques, like path orders or interpretations are employed. All these methods focus on the analysis of the given defined symbols (like for instance the application symbol in the example above) and fail if their recursive definition is too complicated. Naturally this calls for a special treatment of the applicative structure of the system [9].

How could we get rid of those (`@`), thus highlighting the deep recursive structure of the program above? Let us, for example, focus on the rewriting rule $\mathtt{C1}(f,g) \mathbin{@} z \to f \mathbin{@} (g \mathbin{@} z)$, which is particularly nasty for FOPs, given that the variables f and g will be substituted by unknown closure terms, which could potentially be the cause of a very high complexity. How could we *simplify* all this? The key observation is that although this rule tells us how to compose two *arbitrary* closures, only very few instances of the rule above are needed, namely those were g is of the form $\mathtt{C3}(x)$, and f is either $\mathtt{C2}$ or of the form $\mathtt{C1}(f,g)$. This crucial information can be retrieved in the so-called *collecting semantics* [15] of the term rewrite system above, which precisely tells us which objects will possibly be substituted for rule variables along the evaluation of certain families of terms. Dealing with all this fully automatically is of course impossible, but techniques based on tree automata, and inspired by those in [12] can indeed be of help.

Another useful observation is the following: functions like, e.g., `Comp` or $\mathtt{match}_{\mathtt{w}}$ are essentially useless: their only purpose is to build intermediate closures, or to control program flow: one could simply shortcircuit them, using a form of *inlining*. After this is done, some of those rules are *dead code*, and can thus be eliminated from the program. At the end of the day, we arrive at a truly first-order system and *uncurrying* brings it to a format most suitable for FOPs.

If we carefully apply the just described ideas to the example above, we end up with the following first-order system, called $\mathcal{R}_{\mathsf{rev}}$, which is precisely what HoCA produces in output:

Table 1: Experimental Evaluation conducted with TCT and T$_{\text{T}}$T$_2$ on 25 relevant examples.

Inferred Runtime	transformation	#yes	Execution Time (*min* / *avg* / *max*) HoCA	FOP
constant	defunctionalisation	2	—	0.37 / 1.71 / 3.05
	+ simplification	2	0.01 / 2.28 / 4.56	0.23 / 0.51 / 0.79
linear	defunctionalisation	5	—	0.37 / 4.82 / 13.85
	+ simplification	14	0.01 / 0.54 / 4.56	0.23 / 2.53 / 14.00
quadratic	defunctionalisation	5	—	0.37 / 4.82 / 13.85
	+ simplification	18	0.01 / 0.43 / 4.56	0.23 / 6.30 / 30.12
polynomial	defunctionalisation	5	—	0.37 / 4.82 / 13.85
	+ simplification	20	0.01 / 0.42 / 4.56	0.23 / 10.94 / 60.10
terminating	defunctionalisation	8	—	0.83 / 1.38 / 1.87
	+ simplification	25	0.01 / 0.87 / 6.48	0.72 / 1.43 / 3.43

(12) $\texttt{main}(l) \to []$ *(13)* $\texttt{main}(x{::}ys) \to \texttt{C1}^1(\texttt{fix}^1_{\texttt{w}}(ys), \texttt{C3}(x), [])$

(14) $\texttt{fix}^1_{\texttt{w}}([]) \to \texttt{C2}$ *(15)* $\texttt{fix}^1_{\texttt{w}}(x{::}ys) \to \texttt{C1}(\texttt{fix}^1_{\texttt{w}}(ys), \texttt{C3}(x))$

(16) $\texttt{C1}^1(\texttt{C2}, \texttt{C3}(x), z) \to x{::}z$ *(17)* $\texttt{C1}^1(\texttt{C1}(f, g), \texttt{C3}(x), z) \to \texttt{C1}^1(f, g, x{::}z)$.

This term rewrite system is equivalent to $\mathcal{A}_{\mathsf{rev}}$ from above, both extensionally and in terms of the underlying complexity up to speedup by a constant factor. However, the FOPs we have considered can indeed conclude that `main` has linear complexity, a result that can then be lifted back to the original program.

3 Experimental Evaluation

We compiled a diverse collection of higher-order programs from the literature [6, 13, 17] and standard textbooks [5, 19], on which we performed tests with our tool in conjunction with the general-purpose first-order resource analysis tool TCT [3], version 2.1. For comparison, we have also paired HoCA with the termination tool T$_{\text{T}}$T$_2$ [14], version 1.15.

In Table 1 we summarise our experimental findings on the 25 examples from our collection.[4] Rows + simplification in the table indicates the total number of higher-order programs whose runtime could be classified linear, quadratic and at most polynomial when HoCA is paired with the back-end TCT, and those programs that can be shown terminating when HoCA is paired with T$_{\text{T}}$T$_2$. In contrast, row defunctionalisation shows the same statistics when the FOP is run directly on the defunctionalised program. To each of those results, we state the minimum, average and maximum execution time of HoCA and the employed FOP. All experiments were conducted on a machine with an 8 dual-core AMD Opteron™ 885 processors running at 2.60GHz, and 64Gb of RAM.[5] Furthermore, the tools were advised to search for a certificate within 60 seconds. Not all examples in the testbed are subject to a runtime complexity analysis through our approach. However, at least termination can be automatically verified. For all but one example (namely `mapplus.fp`) the obtained complexity certificate is asymptotically optimal.

[4] Examples and full experimental evidence can be found on the HoCA homepage.
[5] Average PassMark CPU Mark 2851; http://www.cpubenchmark.net/.

References

[1] D. Aspinall, L. Beringer, M. Hofmann, H-W. Loidl, and A. Momigliano. A Program Logic for Resources. *TCS*, 389(3):411–445, 2007.

[2] M. Avanzini, U. Dal Lago, and G. Moser. Analysing the complexity of functional programs: higher-order meets first-order. In *Proc. of 20th ICFP*, pages 152–164, 2015.

[3] M. Avanzini and G. Moser. Tyrolean Complexity Tool: Features and Usage. In *Proc. of 24th RTA*, volume 21 of *LIPIcs*, pages 71–80, 2013.

[4] M. Avanzini and G. Moser. A Combination Framework for Complexity. *IC*, 2015. To appear.

[5] R. Bird. *Introduction to Functional Programming using Haskell, Second Edition.* Prentice Hall, 1998.

[6] O Danvy and L. R. Nielsen. Defunctionalization at Work. In *Proc. of 3rd PPDP*, pages 162–174. ACM, 2001.

[7] J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving Termination of Programs Automatically with AProVE. In *Proc. of 7th IJCAR*, volume 8562 of *LNCS*, pages 184–191, 2014.

[8] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated Termination Proofs for Haskell by Term Rewriting. *TOPLAS*, 33(2), 2011.

[9] N. Hirokawa, A. Middeldorp, and H. Zankl. Uncurrying for Termination and Complexity. *JAR*, 50(3):279–315, 2013.

[10] N. Hirokawa and G. Moser. Automated Complexity Analysis Based on the Dependency Pair Method. In *Proc. of 4th IJCAR*, volume 5195 of *LNAI*, pages 364–380, 2008.

[11] N. Hirokawa and G. Moser. Complexity, Graphs, and the Dependency Pair Method. In *Proc. of 15th LPAR*, volume 5330 of *LNCS*, pages 652–666, 2008.

[12] N. D. Jones. Flow Analysis of Lazy Higher-order Functional Programs. *TCS*, 375(1-3):120–136, 2007.

[13] S. Jost, K. Hammond, H.-W. Loidl, and M.Hofmann. Static Determination of Quantitative Resource Usage for Higher-order Programs. In *Proc. of 37th POPL*, pages 223–236. ACM, 2010.

[14] M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2. In *Proc. of 20th RTA*, volume 5595 of *LNCS*, pages 295–304, 2009.

[15] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer, 2005.

[16] L. Noschinski, F. Emmes, and J. Giesl. A Dependency Pair Framework for Innermost Complexity Analysis of Term Rewrite Systems. In *Proc. of 23rd CADE*, LNAI, pages 422–438. Springer, 2011.

[17] C. Okasaki. Functional Pearl: Even Higher-Order Functions for Parsing. *JFP*, 8(2):195–199, 1998.

[18] S. E. Panitz and M. Schmidt-Schauß. TEA: Automatically Proving Termination of Programs in a Non-Strict Higher-Order Functional Language. In *Proc. of 4th SAS*, pages 345–360, 1997.

[19] F. Rabhi and G. Lapalme. *Algorithms: A Functional Programming Approach.* Addison-Wesley, 1999.

[20] J. C. Reynolds. Definitional Interpreters for Higher-Order Programming Languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

[21] M. Sinn, F. Zuleger, and H. Veith. A Simple and Scalable Static Analysis for Bound Analysis and Amortized Complexity Analysis. In *Proc. of 26th CAV*, volume 8559 of *LNCS*, pages 745–761, 2014.

[22] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenstrom. The Worst Case Execution Time Problem - Overview of Methods and Survey of Tools. *TECS*, 2008.

Degrees of extensionality in the theory of Böhm trees

Jörg Endrullis

Vrije Universiteit Amsterdam
The Netherlands
j.endrullis@vu.nl

Abstract

We will discuss the clocked lambda-calculus, an extension of the classical lambda-calculus. This extension is infinitary strongly normalizing, infinitary confluent, and the unique infinitary normal forms constitute enriched Böhm Trees, which we call clocked Böhm Trees. These are suitable for discriminating a rich class of lambda-terms having the same Böhm Trees.

Degrees of extensionality in the theory of Böhm trees

Giulio Manzonetto

Université Paris 13, Sorbonne Paris Cité, F-93430,
Villetaneuse, France
giulio.manzonetto@lipn.univ-paris3.fr

Abstract

The problem of determining when two programs are equivalent is central in computer science, e.g., it is necessary to verify that the optimizations performed by a compiler actually preserve the meaning of the input program. For λ-calculi, Morris proposed in his thesis [5] to consider two λ-terms M and N as equivalent when they are *contextually equivalent* with respect to some fixed set $\mathcal{O}$ of *observables*. Let us call $\mathcal{T}_{\mathcal{O}}$ the observational theory with observables $\mathcal{O}$:

$$\mathcal{T}_{\mathcal{O}} \vdash M = N \iff \forall C[] \,.\, [\, C[M] \in \mathcal{O} \iff C[N] \in \mathcal{O} \,]$$

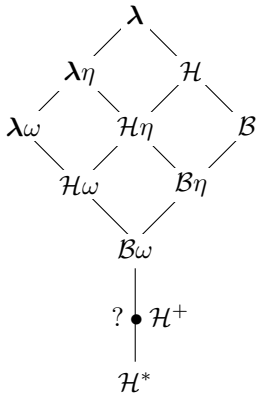
The problem of this definition is that the quantification over all possible contexts is difficult to handle in practice. Therefore, many researchers over the years undertook a quest for characterizing observational equivalences both semantically, by defining fully abstract denotational models, and syntactically, by introducing several kinds of extensional equivalences on Böhm trees (that are possibly infinite trees representing program executions).

The observational theory $\mathcal{H}^* := \mathcal{T}_{\mathrm{SOL}}$ where the observables are the solvable (equivalently, head-normalizable) λ-terms is by far the most well studied theory of λ-calculus — it is the theory of Scott's $\mathcal{D}_\infty$ model [6] and equates two λ-terms exactly when their Böhm trees are equal up to possibly infinite η-expansions. Curiously, the first extensional observational theory that has been defined in the literature is not $\mathcal{H}^*$, but rather $\mathcal{H}^+ := \mathcal{T}_{\mathrm{NF}}$ where the observables are the β-normalizable λ-terms. This theory has been little studied in the literature, but a fully abstract filter model has been defined by Coppo *et al.* in [3] and it is well-known that two λ-terms are equal in $\mathcal{H}^+$ whenever their Böhm trees coincide up to finite η-expansions.

It should now be clear that observational theories and extensional equivalences are tightly connected. Now, the λ-calculus admits a notion of extensionality a priori stronger than η:

$$(\omega\text{-rule}) \qquad \forall \text{ closed } \lambda\text{-term } P \,.\, MP = NP \quad \Longrightarrow \quad M = N$$

so, it is natural to wonder how this rule compares with the notions of extensionality above. It is well-known that neither $\boldsymbol{\lambda}$ (the theory of β-equivalence), $\mathcal{H}$ (the least theory equating all unsolvables), $\mathcal{B}$ (the theory of Böhm trees) nor their extensional versions $\boldsymbol{\lambda}\eta$, $\mathcal{H}\eta$ and $\mathcal{B}\eta$ do satisfy the (ω)-rule. Given a theory $\mathcal{T}$ it is however possible to define $\mathcal{T}\omega$ as the least theory satisfying the ω-rule.
In Barendregt's book [1] a "kite" shaped diagram depicts all strict inclusion relations among these theories (see the figure on the right, where $\mathcal{T}_1$ is above $\mathcal{T}_2$ whenever $\mathcal{T}_1 \subsetneq \mathcal{T}_2$). In the seventies Barendregt raised the question of determining the position of $\mathcal{H}^+$ in this diagram, and in 1978 Sallé conjectured that $\mathcal{B}\omega \subsetneq \mathcal{H}^+$. The problem remained open for almost 40 years. In 2016 Breuvart *et al.* proved that $\mathcal{B}\omega \subseteq \mathcal{H}^+$ [2]. Sallé's conjecture has been refuted by Intrigila *et al.* in 2017 by showing that $\mathcal{B}\omega$ and $\mathcal{H}^+$ actually coincide [4]. As a byproduct, we obtain a characterization of all degrees of extensionality in $\mathcal{B}$.



References

[1] H. P. Barendregt. *The lambda-calculus, its syntax and semantics*. Number 103 in Studies in Logic and the Foundations of Mathematics. revised edition, 1984.

[2] Flavien Breuvart, Giulio Manzonetto, Andrew Polonsky, and Domenico Ruoppolo. New results on Morris's observational theory: The benefits of separating the inseparable. In Delia Kesner and Brigitte Pientka, editors, *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal*, volume 52 of *LIPIcs*, pages 15:1–15:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2016.

[3] Mario Coppo, Mariangiola Dezani-Ciancaglini, and Maddalena Zacchi. Type theories, normal forms and $\mathcal{D}_\infty$-lambda-models. *Inf. Comput.*, 72(2):85–116, 1987.

[4] Benedetto Intrigila, Giulio Manzonetto, and Andrew Polonsky. Refutation of Sallé's longstanding conjecture. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction, FSCD 2017, September 3-9, 2017, Oxford, UK*, volume 84 of *LIPIcs*, pages 20:1–20:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[5] J.H. Morris. *Lambda calculus models of programming languages*. Phd thesis, MIT, 1968.

[6] Dana S. Scott. Continuous lattices. In Lawvere, editor, *Toposes, Algebraic Geometry and Logic*, volume 274 of *Lecture Notes in Mathematics*, pages 97–136. Springer, 1972.

SizeChangeTool: A Termination Checker for Rewriting Dependent Types

Guillaume Genestier[1,2]

[1] LSV, ENS Paris-Saclay, CNRS, Université Paris-Saclay
[2] MINES ParisTech, PSL University

1 Introduction

SizeChangeTool [10] is a fully automated termination checker for the $\lambda\Pi$-calculus modulo rewriting. Its development became essential as various libraries were encoded in an implementation of this logic: the logical framework Dedukti [3].

A logical framework allows the user to define the logic they want to reason with and then use it to actually write proofs. To define a logic in Dedukti, the user provides a set of rewriting rules. Those rules do not only define functions, but can also define types. However, to ensure that the defined type system has good properties, like logical consistency or decidability, the rules must satisfy some properties: termination, confluence and type preservation.

Many criteria have been created to check termination of first-order rewriting. For instance, dependency pairs [2], which evolved in a complete framework [21] or size-change termination [18], just to mention those appearing in this work. The dynamism of this research area is illustrated by the numerous tools participating in the various first-order categories of the termination competition [20]. For higher-order rewriting too, criteria have been crafted, many of them can be found in [15] and a category exists in the competition. However, one can deplore the small number of participants in this category: Only 2 in 2019, including SizeChangeTool!

This lack of implementations is even more visible for rewriting with dependent types, for which criteria have been developed [5, 14], but as far as the author knows, none of them have been implemented.

Outline After presenting the logical system and examples of programs in Sec. 2, we present the criterion used by the tool in Sec. 3. Sec. 4 details the implementation choices of SizeChangeTool and Sec. 5 compares it with the others termination checkers.

2 The $\lambda\Pi$-calculus Modulo Rewriting

The $\lambda\Pi$-calculus modulo rewriting ($\lambda\Pi/\mathcal{R}$ for short) is an extension of the logical framework LF [12]. It is a system of dependent types where types are identified not only modulo the β-conversion of λ-calculus, but also by user-given rewriting rules.

Definition 1. *$\lambda\Pi/\mathcal{R}$ extends the Pure Type System λP [4] with a finite signature $\mathbb{F}$ and a set of rules $\mathcal{R} = (\Delta, f\,\bar{l} \to r)$ such that $f \in \mathbb{F}$, $\mathrm{FV}(r) \subseteq \mathrm{FV}(\bar{l})$ and Δ is a context associating a type to every variable of $\bar{l}$. $\to_{\mathcal{R}}$ is the closure by substitution and context of $\mathcal{R}$.*

The conversion rule is enriched to take into account rewriting rules:

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad A \leftrightarrow^*_{\beta\mathcal{R}} B}{\Gamma \vdash t : B}\ (conv)$$

Note that the constraints on the rewriting rules are very loose. In particular, we do not enforce the rules to be orthogonal, meaning that overlapping or non-linear rules are allowed. Let us give two examples, highlighting the possibilities offered by the system. A more comprehensive example can be found in [7].

Example 2 (Summation of variable arity)**.** *Rewriting rules at type level allow us for instance to define* `F n` *as the type* `Nat ⇒ Nat ⇒...⇒ Nat` *with* n *arrows. With it, we can type the function* `sum` *which is such that* `sum n` $l_1 \ldots l_n$ `=` l_1 `+...+` l_n[1]*.*

```
symbol Nat : TYPE
symbol const 0 : Nat                    symbol const s : Nat ⇒ Nat

symbol plus : Nat ⇒ Nat ⇒ Nat           set infix "+" ≔ plus
  rule 0 + &y → &y                      rule (s &x) + &y → s (&x + &y)

symbol F : Nat ⇒ TYPE
  rule F 0 → Nat                        rule F (s &n) → Nat ⇒ F &n

symbol sum : ∀ n: Nat, F n
  rule sum 0           → 0              rule sum (s 0) → λx, x
  rule sum (s (s &n)) → λx y, sum (s &n) (x + y)
```

Example 3 (Simply-typed λ-calculus)**.** *A simple instance of encoding of logic in* Dedukti *is the simply-typed* λ*-calculus, which is presented here with the type* `typ` *for code of types and* `T` *which decodes an element of* `typ` *into a type of* Dedukti*.*

```
symbol typ : TYPE              symbol arrow : typ ⇒ typ ⇒ typ
symbol T : typ ⇒ TYPE

symbol lambda : ∀(a b : typ), (T a ⇒ T b) ⇒ T (arrow a b)
symbol appli  : ∀(a b : typ), T (arrow a b) ⇒ T a ⇒ T b
  rule appli &a &b (lambda _ _ &f) &x → &f &x
```

We are interested in the strong normalization of $\rightarrow_{\beta\mathcal{R}} = (\rightarrow_\beta \cup \rightarrow_\mathcal{R})$.

3 Dependency Pairs and Size-Change Termination

Dependency pairs are at the core of all the state-of-the-art automated termination provers for first-order term rewriting systems. Arts and Giesl [2] proved that a first-order rewriting relation is terminating if and only if there is no infinite chain, that is sequence of dependency pairs interleaved with reductions in the arguments. This notion of dependency pair has been extended to higher order [6, 9], however those extensions do not include dependent types, which is a compulsory feature when we are developing a logical framework.

Definition 4 (Dependency pairs)**.** *Let* $f\,\bar{l} > g\,\bar{m}$ *iff there is a rule* $f\,\bar{l} \rightarrow r \in \mathcal{R}$*,* g *is the head of the left-hand side of a rule and* $g\,\bar{m}$ *is a subterm of* r *maximally applied.*

$f\,t_1 \ldots t_p \stackrel{\sim}{>} g\,u_1 \ldots u_q$ *iff there are a dependency pair* $f\,l_1 \ldots l_i > g\,m_1 \ldots m_j$ *with* $i \leqslant p$ *and* $j \leqslant q$ *and a substitution* σ *such that, for all* $k \leqslant i$*,* $t_k \rightarrow^*_{\beta\mathcal{R}} l_k\sigma$ *and, for all* $k \leqslant j$*,* $m_k\sigma = u_k$*.*

[1] & is used in Dedukti to identify pattern variables in rewriting rules.

One criterion for first-order rewriting is Lee, Jones and Ben-Amram size-change termination criterion (SCT) [18]. It consists in following the arguments through sequences of recursive calls and checking that, in every potential loop, one of them strictly decreases.

Definition 5 (Size-Change Termination)**.** *Let $\triangleright$ be a well-founded order on terms. The call graph $\mathcal{G}(\mathcal{R}, \triangleright)$ associated to $\mathcal{R}$ is the directed labeled graph on the symbols of $\mathbb{F}$ such that there is an edge between f and g iff there is a dependency pair $f\, l_1 \dots l_p > g\, m_1 \dots m_q$. This edge is labeled with the matrix $(a_{i,j})_{i \leq ar(f), j \leq ar(g)}$ where:*

- *if $l_i \triangleright m_j$, then $a_{i,j} = -1$;*
- *if $l_i = m_j$, then $a_{i,j} = 0$;*
- *otherwise $a_{i,j} = \infty$ (in particular if $i > p$ or $j > q$).*

$\mathcal{R}$ is size-change terminating for $\triangleright$ if, in the transitive closure of $\mathcal{G}(\mathcal{R}, \triangleright)$ (using the min-plus semi-ring to multiply the matrices labeling the edges), all idempotent matrices labeling a loop have some -1 on their diagonal.

In [7], we present an adaptation of dependency pairs to $\lambda\Pi/\mathcal{R}$ and prove that (under some conditions) the absence of infinite chains implies the termination of $\rightarrow_{\beta\mathcal{R}}$. After Wahlstedt [22], we used an adaptation of SCT to check the absence of infinite chains of dependency pairs.

Definition 6 (Well-Structured System)**.** *We consider a pre-order $\succeq$ on $\mathbb{F}$ such that if g occurs in the type of f or in the right-hand side of a rewriting rule defining f, then $f \succeq g$. $\mathcal{R}$ is well-structured if for every rule $(\Delta, f\, \bar{l} \rightarrow r)$, if f is of type $\Pi(\bar{x} : \bar{T}).U$, then $\Delta \vdash r : U[\bar{x} \rightarrow \bar{l}]$ is derivable using only symbols smaller or equal to f.*

The result of [7] is:

Theorem 7. *The relation $\rightarrow_{\beta\mathcal{R}}$ terminates on terms typable in $\lambda\Pi/\mathcal{R}$ if $\rightarrow_{\beta\mathcal{R}}$ is locally confluent and preserves typing, $\mathcal{R}$ is well-structured, size-change terminating for the subterm ordering and plain-function passing.*

where "plain-function passing" is a quite restrictive condition on the variable allowed to occur in the right-hand side of rules.

In SizeChangeTool, the criterion used is a (still unpublished) extension of this result where we replace the plain-function passing hypothesis by a condition analogous to strict positivity of inductive types and use the structural ordering introduced in [8] for checking size-change termination.

Extension 8. *The relation $\rightarrow_{\beta\mathcal{R}}$ terminates on terms typable in $\lambda\Pi/\mathcal{R}$ if $\rightarrow_{\beta\mathcal{R}}$ is locally confluent and preserves typing, $\mathcal{R}$ is well-structured, is size-change terminating for the structural ordering and there is a pre-order between types such that for every rule $(\Delta, f\, \bar{l} \rightarrow r)$ and every $c \in \mathbb{F}$ occurring in a l_i, the type of c is strictly positive for this pre-order.*

4 Implementation and interaction with the type-checker

SizeChangeTool takes as input Dedukti files or XTC files, the format of the termination competition [20]. However, XTC does not include dependent types now, hence we proposed a backward compatible extension of the format. In fact, the tool accepts this format extension.

Checking that the provided rules are confluent with β is left to the user. To check it automatically, Dedukti offers an export to the format of the confluence competition. SizeChangeTool performs check of the 4 remaining hypotheses, to use the extension 8 of Thm. 7.

1. *type preservation* is checked by Dedukti assuming that the provided rewrite rules are confluent with β. This algorithm can be found in [3].

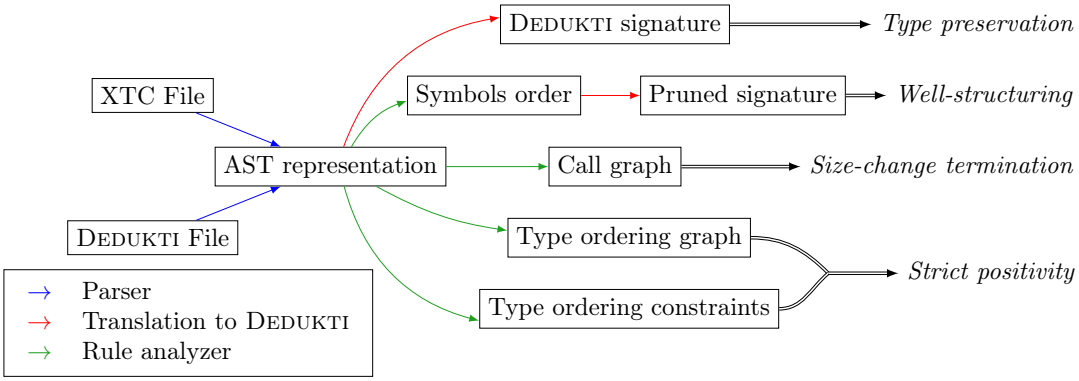


Figure 1: SizeChangeTool Workflow

2. *well-structuring* requires to construct the pre-order described in Def. 6. Once this pre-order is computed, Dedukti is reused to check if it is possible to type the right-hand side of every rule using the pruned signature where symbols greater than the one defined are removed.

3. *size-change termination* requires to analyze every rule in order to extract the dependency pairs. Then the call-graph is constructed. To perform size-change termination checking, one must compute the transitive closure of the call graph and verify the presence of a -1 on the diagonal of every idempotent matrix labeling a loop. This check has been implemented by Lepigre and Raffalli for the language PML_2 [19]. SizeChangeTool reuses their work to analyze the call graph.

4. *the strict positivity condition* requires to have a pre-order on type constructors. The user is not asked to provide this order. While analyzing the rules, SizeChangeTool constructs a graph whose vertices are type constructors and arrows means "is smaller or equal to" as well as a list of constraints of the form "Type constructor A is strictly greater than type constructor B." To check that this relation is a pre-order, one checks that for every constraint "A is strictly smaller than B." there is no arrow between A and B in the transitive closure of the graph.

For the sake of simplicity, representation of terms and rules are mainly shared between Dedukti and SizeChangeTool. So the red arrows on Fig. 1 are (almost) the identity. However those translation functions are made explicit, since one could imagine plugging another type-checker on the rule analyzer offered by SizeChangeTool.

5 Comparison with other tools

As far as the author knows, there are no other termination checker combining dependent types and non-orthogonal rewriting rules. However, dropping one of these features and restricting ourselves to simply-typed higher-order rewriting systems or to dependently-typed orthogonal systems permits comparison with existing tools.

For simply-typed systems, the termination competition [20] proposes a category "higher-order rewriting union beta". In 2019, there were only two tools competing in this category: SizeChangeTool and Wanda [17]. Wanda uses multiple techniques to prove termination:

dependency pairs, polynomial interpretations, HORPO... [15]. Unsurprisingly, the sole criterion used in SizeChangeTool cannot prove as many examples as this wide range of techniques.

However, on the bench of the competition, SizeChangeTool is 11 times faster than Wanda. The speed of SizeChangeTool permits it to show in less than 0.1 second termination of 3 examples on which Wanda is unable to answer with a timeout of 300 seconds: `Mixed_HO_10/deriv.xml` encodes derivation of usual mathematical functions, like:[2]

```
rule der (λx, (&F x) + (&G x)) → λx: real, (der &F x) + (der &G x)
rule der (λx, ln (&F x))       → λx: real, (der &F x) / (&F x)
```

`Hamana_17/churchNum.xml` and `Hamana_17/churchNum2.xml`, contain the Church encoding of natural numbers, with rules like:

```
rule two (λx, &I x) (λx, &J x) (λx, &F1 x) &Y1
     → &I (&I (λy, &J y)) (λy, &F1 y) &Y1
```

The very low time consumption of the presented criterion suggests that Wanda would improve significantly its efficiency by implementing this technique.

If we restrict ourselves to orthogonal systems, it is then possible to compare our technique to the ones implemented in Coq and Agda. Coq essentially implements a higher-order version of primitive recursion [11], whereas Agda uses subterm criterion (a criterion very similar to size-change termination) [1]. Hence, Coq cannot handle function definitions with permuted arguments in function calls, which is not a problem for Agda and SizeChangeTool. Agda recently added the possibility of declaring rewriting rules but this feature is highly experimental and no check is performed on the rules. In particular, Agda termination checker does not handle rewriting rules.

6 Conclusion and future work

For now on, the accepted input files are restricted to Dedukti and XTC files. One could imagine extending it to other input formats, for instance the rewriting rules offered in Agda.

Following the approach adopted by Wanda, one could also just study truly higher-order rules, use a state-of-the-art first-order prover for the remaining rules and then rely on a modularity theorem to conclude. This strategy would improve the performance of SizeChangeTool in the competition, since, according to C. Kop: "about half the benchmarks now do little more than test the strength of the first-order back-end that some higher-order tools use." [16].

One could also think of various enhancement of the criterion, for instance to handle rules with a local increase of the size of the arguments like in:

```
rule f &x → g (s &x)          rule g (s (s &x)) → f &x
```

Hyvernat proposed such an extension of SCT for constructor-based first-order languages [13].

Adapting other so-called "dependency pairs processors" [9] to the $\lambda\Pi/\mathcal{R}$ is of course another active subject of study and would improve the tool.

Now that a higher-order rewriting with dependent types termination prover has been developed, one can hope such development will emulate other researches. The adoption of an extension of XTC and the creation of a category for $\lambda\Pi/\mathcal{R}$ in the competition, would probably support the creation of such new implementations.

[2] For sake of readability, examples are presented in Dedukti syntax and some η-reduction are performed.

References

[1] A. Abel. foetus – Termination Checker for Simple Functional Programs. 1998.

[2] T. Arts, J. Giesl. Termination of term rewriting using dependency pairs. *TCS* 236:133–178, 2000.

[3] A. Assaf, G. Burel, R. Cauderlier, D. Delahaye, G. Dowek, C. Dubois, F. Gilbert, P. Halmagrand, O. Hermant, R. Saillard. Dedukti: a Logical Framework based on the λΠ-Calculus Modulo Theory.

[4] H. Barendregt. Lambda calculi with types. In *Handbook of logic in computer science. Volume 2. Background: computational structures*, p. 117–309. Oxford University Press, 1992.

[5] F. Blanqui. Definitions by rewriting in the calculus of constructions. *MSCS* 15(1):37–92, 2005.

[6] F. Blanqui. Higher-order dependency pairs. WST, 2006.

[7] F. Blanqui, G. Genestier, O. Hermant. Dependency Pairs Termination in Dependent Type Theory Modulo Rewriting. FSCD, 2019.

[8] T. Coquand. Pattern matching with dependent types. TYPES, 1992.

[9] C. Fuhs, C. Kop. A Static Higher-Order Dependency Pair Framework. ESOP, LNCS, 2019.

[10] G. Genestier. SizeChangeTool. `https://github.com/Deducteam/SizeChangeTool`, 2018.

[11] E. Giménez. Codifying Guarded Definitions with Recursive Schemes. TYPES 1994.

[12] R. Harper, F. Honsell, G. Plotkin. A framework for defining logics. *JACM* 40(1):143–184, 1993.

[13] P. Hyvernat. The size-change termination principle for constructor based languages. *LMCS* 2014.

[14] J.-P. Jouannaud, J. Li. Termination of Dependently Typed Rewrite Rules. TLCA, 2015.

[15] C. Kop. *Higher order termination*. PhD thesis, VU University Amsterdam, 2012.

[16] C. Kop. Mail to the termtools list. higher-order union beta category in the TPDB. 19/03/2019.

[17] C. Kop. Wanda. `http://wandahot.sourceforge.net/`.

[18] C. S. Lee, N. Jones, A. Ben-Amram. The size-change principle for program termination. POPL'01.

[19] R. Lepigre. PML_2. `https://github.com/rlepigre/pml`, 2017.

[20] Termination Competition. `http://termination-portal.org/wiki/Termination_Competition`.

[21] R. Thiemann. *The DP framework for proving termination of term rewriting*. PhD thesis, RWTH Aachen University, 2007. Technical Report AIB-2007-17.

[22] D. Wahlstedt. *Dependent type theory with first-order parameterized data types and well-founded recursion*. PhD thesis, Chalmers University of Technology, 2007.

IWC 2019

A short overview of Wanda

Cynthia Kop[1]

Radboud University Nijmegen, Netherlands
C.Kop@cs.ru.nl

Abstract

Wanda is a fully automatic termination analysis tool for higher-order term rewriting. In this paper, we will discuss Wanda's underlying methodology. Most pertinently, this includes a higher-order dependency pair framework, weakly monotonic algebras through higher-order polynomials, and a variation of the higher-order recursive path ordering. All techniques are employed automatically using SAT encodings.

1 Introduction

Termination of term rewriting systems (TRSs) has been an area of active research for several decades. In the last twenty years the field of *automatically* proving termination has flourished, and several strong provers have been developed to participate against each other in the annual *Termination Competition* [22]. This competition includes various categories, including one for *higher-order* term rewriting. This area of rewriting presents some unique challenges, for example due to bound variables. Nevertheless, several tools have participated in this category (Hot [2]), `THOR` [6], `SizeChangeTool`, `Sol` [12], Wanda), each using different techniques.

Wanda, a tool built primarily around *higher-order dependency pairs*, has been the leading tool in this category since 2013. Wanda has also been used as a termination back-end in the higher-order category of the *International Confluence Competition* [7], with both participating tools in 2016 (ACPH [18] and CSIˆho [17]) delegating termination questions to Wanda.

In this paper I will discuss the most important techniques used in Wanda. To this end I follow roughly the structure of an analysis by Wanda: first a higher-order TRS is read and (if necessary) translated into Wanda's internal formalism, *AFSMs* (§2); then basic techniques for non-termination (§3) and for simple termination proofs using reduction pairs (§4) are applied. Finally, responsibility is passed to the dependency pair framework (§5).

2 Higher-order term rewriting using AFSMs

Unlike first-order term rewriting, there is no single, unified approach to higher-order term rewriting, but rather a number of similar but not fully compatible systems aiming to combine term rewriting and typed λ-calculi. This is a problem, since users of various kinds of higher-order TRSs may be interested in termination, and it would be frustrating to write a tool with slightly altered techniques for every single style. Therefore, Wanda uses her own internal format, *AFSMs*, which several popular kinds of rewriting systems can be translated into. AFSMs (Algebraic Functional Systems with Meta-variables) are essentially simply-typed CRSs [13] and also largely correspond to the formalism in [4]; they are fully explained in [15, Ch. 2] and in [10].

Terms are built from a set of simply-typed variables $\mathcal{V}$ and a set $\mathcal{F}$ of simply-typed function symbols, using abstraction and application to form well-typed expressions. Term equality is modulo α-conversion, but not modulo β or η. *Meta-terms* may additionally use *meta-variable applications*, which are essentially unbound variables applied to a fixed number of arguments. *Rules* are pairs $\ell \Rightarrow r$ of meta-terms of the same type, such that all meta-variables in r occur also in ℓ, both sides are *closed* (all their variable occurrences are bound) and ℓ is a *pattern*:

for all sub-meta-terms of ℓ which have the form $Z[s_1, \ldots, s_k]\ t_1 \cdots t_n$ necessarily $n = 0$ and $s_1, \ldots, s_k$ are distinct variables. A set of rules $\mathcal{R}$ defines a *rewrite relation* $\Rightarrow_{\mathcal{R}}$ as the smallest monotonic relation on terms that contains all instances of the rules as well as beta-reduction.

Meta-variables are used in early forms of higher-order rewriting such as Aczel's Contraction Schemes [1] and Klop's Combinatory Reduction Systems [13]. They strike a balance between matching modulo β-reduction and syntactic matching.

Example 2.1. The common example of a `map` function that applies a function on all elements of a list can be expressed in two ways. First, with meta-variables that do not take arguments:

$$\begin{array}{rcl} \texttt{map}\ F\ \texttt{nil} & \Rightarrow & \texttt{nil} \\ \texttt{map}\ F\ (\texttt{cons}\ H\ T) & \Rightarrow & \texttt{cons}\ (F\ H)\ (\texttt{map}\ F\ T) \end{array}$$

Second, with the meta-variable F of higher type taking one argument:

$$\begin{array}{rcl} \texttt{map}\ (\lambda x.F[x])\ \texttt{nil} & \Rightarrow & \texttt{nil} \\ \texttt{map}\ (\lambda x.F[x])\ (\texttt{cons}\ H\ T) & \Rightarrow & \texttt{cons}\ F[H]\ (\texttt{map}\ (\lambda x.F[x])\ T) \end{array}$$

Both have similar typing and termination behaviour. However, note that $\lambda x.F[x]$ only matches an abstraction: with the former rules, `map suc nil` can be rewritten; with the latter it cannot.

Example 2.2. Meta-variables are necessary to express certain rules; for example the rule $\texttt{deriv}\ (\lambda x.\texttt{sin}\ F[x]) \Rightarrow \lambda y.\texttt{times}\ (\texttt{deriv}\ (\lambda x.F[x])\ y)\ (\texttt{cos}\ F[y])$. Here, we cannot replace $F[x]$ by an application $F\ x$ because the result would not be a pattern (and would actually have very different matching behaviour). Meta-variables with arguments can also alter termination behaviour: an AFSM with rules $\texttt{a} \Rightarrow \texttt{f}\ (\lambda x.\texttt{b})$ and $\texttt{f}\ (\lambda x.F[x]) \Rightarrow F[\texttt{a}]$ is terminating (since $\texttt{a} \Rightarrow_{\mathcal{R}} \texttt{f}\ (\lambda x.\texttt{b}) \Rightarrow_{\mathcal{R}} \texttt{b}$: the application of the abstraction to `a` is immediately evaluated), but we obtain non-termination if the second rule is replaced by $\texttt{f}\ F \Rightarrow F\ \texttt{a}$ (since then $a \Rightarrow_{\mathcal{R}} \texttt{f}\ (\lambda x.\texttt{b}) \Rightarrow_{\mathcal{R}} (\lambda x.\texttt{b})\ \texttt{a}$, and we are not obliged to immediately β-reduce this term).

Various styles of higher-order rewriting use *functional* notation, e.g., $\texttt{map}(F, \texttt{cons}(H, T))$. This is not merely a notational difference: in such a system, `map` cannot occur with only 1 argument. However, following [15, §2.3.1] and [14, §7], *uncurrying* does not affect termination provided the rules are (essentially) unchanged: if a symbol `f` is always applied to at least k arguments in the rules (i.e., `f` always occurs in the form $\texttt{f}\ s_1 \cdots s_n$ with $n \geq k$) then we may assign `f` an "arity" of k arguments (i.e., replace the above by $\texttt{f}(s_1, \ldots, s_k)\ s_{k+1} \cdots s_n$). For this reason, Wanda only allows applicative notation in the input, but then *derives* an arity and uses functional notation in the output. To illustrate, in Example 2.1 `map` and `cons` have arity 2, so the second rule would be denoted as $\texttt{map}(F, \texttt{cons}(H, T)) \to \texttt{cons}(F\ H, \texttt{map}(F, T))$.

Input to Wanda. Wanda accepts input files in the format of the termination competition (which uses variables rather than meta-variables for matching and does not satisfy the pattern restriction; this is transformed following [15] into an AFSM), or in a custom AFSM format:

```
nil : list
cons : nat -> list -> list
map : (nat -> nat) -> list -> list

map (/\x:nat.Z[x]) nil => nil
map (/\x:nat.Z[x]) (cons H T) => cons Z[H] (map Z T)
```

Wanda automatically derives arity 2 for both `cons` and `map`. Types should in principle be simple types; type variables (which have a name that starts with $, e.g., `$a`, `$b`) *are* permitted, but most termination techniques will fail for such polymorphic systems.

3 Non-termination

Once the input has been read and – if necessary – converted to AFSM format, it is passed on to the non-termination module. Since the focus in Wanda's development has been on termination, this module is very basic; it does only two things:

- First, right-hand sides of loops are evaluated to detect obvious loops. This is very weak, but may capture systems such as $\{\mathtt{a} \Rightarrow \mathtt{f}\ (\lambda x.\mathtt{b}),\ \mathtt{f}\ F \Rightarrow F\ \mathtt{a}\}$ or $\{\mathtt{f}\ X \Rightarrow \mathtt{f}\ (\mathtt{g}\ X)\}$.
- Second, since β-reduction is included in the rewrite relation, note that the rule $\mathtt{f}\ (\mathtt{g}\ X) \Rightarrow X$ is non-terminating if $\mathtt{g} : (\alpha \to \alpha) \to \alpha$ and $\mathtt{f} : \alpha \to \alpha \to \alpha$ for some type α: we then have, for $\omega := \mathtt{g}\ (\lambda z.\mathtt{f}\ z\ z)$, that $\mathtt{f}\ \omega\ \omega \Rightarrow_{\mathcal{R}} (\lambda z.\mathtt{f}\ z\ z)\ \omega \Rightarrow_{\beta} \mathtt{f}\ \omega\ \omega$. Wanda detects variations of this example, such as encodings of the simply-typed λ-calculus.

Outside this module, non-termination is also shown as part of the DP framework (see §5).

4 Rule removal

Next, control passes to the *rule removal* module, which tries to embed the reduction relation $\Rightarrow_{\mathcal{R}}$ in the union of a *well-founded ordering* $\succ$ and a compatible quasi-ordering $\succsim$. If this succeeds, the rules which cause a decrease by $\succ$ can be deleted.

This module is not *necessary*: reduction pairs $(\succsim, \succ)$ are also used in the dependency pair framework, where the requirements are more liberal. However, a proof using rule removal is often simpler (giving more easily understandable proofs), and faster to find.

Wanda has two ways to generate reduction pairs: higher-order polynomial interpretations and a version of the higher-order recursive path ordering. Both techniques originate from first-order methods, and work best on terms in *functional* notation. Thus, the rules are uncurried, using the "arity" discussed in §2 and new symbols @ to encode application as a function.

Example 4.1. The first AFSM of Example 2.1 has the following equivalent functional notation:

$$\begin{array}{rcll} \mathtt{map}(F, \mathtt{nil}) & \Rightarrow & \mathtt{nil} & \\ \mathtt{map}(F, \mathtt{cons}(H, T)) & \Rightarrow & \mathtt{cons}(@^{\mathtt{nat},\mathtt{nat}}(F, H), \mathtt{map}(F, T)) & (\text{where } F : \mathtt{nat} \to \mathtt{nat}) \\ @^{\sigma,\tau}(\lambda x.F[x], Z) & \Rightarrow & F[Z] & \text{for all types } \sigma, \tau \end{array}$$

Here, symbols $@^{\sigma,\tau} : (\sigma \to \tau) \to \sigma \to \tau$ and corresponding "beta"-rules are added for all σ, τ. However, these extra rules are ignored in the implementation, as both higher-order polynomial interpretations and StarHorpo are designed so that steps with these rules are always oriented.

Higher-order polynomial interpretations. Wanda's first way to generate a reduction pair is based on v.d. Pol's *weakly monotonic functionals* [19]: all closed terms s are interpreted into "weakly monotonic functionals" $[\![s]\!]$ over the natural numbers, in such a way that $[\![s]\!] \geq [\![t]\!]$ whenever $s \Rightarrow_{\mathcal{R}} t$; if a reduction with a certain rule always gives $[\![s]\!] > [\![t]\!]$, this rule may be removed. For the sake of automation, Wanda particularly considers interpretations into *higher-order polynomials* [9], which is implemented using an encoding to SAT.

Example 4.2. Consider the AFSM of Examples 2.1 and 4.1. The type of `map` is $(\mathtt{nat} \to \mathtt{nat}) \to \mathtt{list} \to \mathtt{list}$, so `map` should be interpreted by a weakly monotonic function that takes as arguments: (1) a weakly monotonic function from $\mathbb{N}$ to $\mathbb{N}$, and (2) a natural number. We do this by choosing $[\![nil]\!] = 0$ and $[\![\mathtt{cons}(s,t)]\!] = 1 + [\![s]\!] + [\![t]\!]$ and $[\![\mathtt{map}(S,t)]\!] = 1 + [\![t]\!] + [\![S]\!](0) + [\![t]\!] * [\![S]\!]([\![t]\!])$ and $[\![@^{\mathtt{nat},\mathtt{nat}}]\!](S,t) = [\![S]\!]([\![t]\!]) + [\![t]\!]$. With this choice, the interpretations of the left-hand sides of the `map` rules are strictly greater than those of the right-hand sides.

StarHorpo The recursive path ordering in Wanda is somewhat similar to the Computability Path Ordering [5], but does not yet consider positive inductive types. On the other hand, Wanda's "StarHorpo" is stronger in some ways due to added features such as minimal symbols and the treatment of application as a function symbol. A peculiarity of StarHorpo compared to CPO is the use of "marked symbols" $\mathtt{f}^\star$: an intermediate term $\mathtt{f}^\star(s_1, \ldots, s_k)$ serves as an upper bound for terms strictly stronger than $\mathtt{f}(s_1, \ldots, s_k)$. This is used to obtain a transitive definition (in contrast to CPO, where formally the ordering is defined as the transitive closure of the definition). The full explanation of StarHorpo is available in [15, Chapter 5].

5 The higher-order dependency pair framework

If any rules remain, Wanda passes them on to the *dependency pair* module, which encompasses the most powerful techniques of the tool. First, this module identifies the largest first-order sub-TRS and passes it (with or without some extra rules) to a first-order termination prover. Following [8], this may both be used to prove non-termination (with some extra checks, since first-order termination tools do not consider typing), or to remove the dependency pairs for this sub-TRS, which significantly lowers the proof obligation for the dependency pair framework.

Essentially, the DP framework within Wanda executes the following algorithm:

1. The AFSM is translated into an initial "DP problem": a set of DPs (essentially: tuples that identify *function* calls in the rules) coupled with a set of rules and some properties.
2. Then, a set S of DP problems (originally just the one above), is iteratively transformed using *DP processors*. If this process reaches $S = \emptyset$, Wanda concludes termination.

Formally, the DP framework [11, 10] can also be used to prove non-termination, but this is not done in Wanda. For ((2), the strongest processors are the *dependency graph* and *reduction triples*: these triples use the orderings from §4, but have more liberal restrictions.

There are two variations of dependency pairs for higher-order rewriting in the literature: *dynamic* [20, 16] and *static* [21, 10], each with distinct advantages and disadvantages. Both of them are adapted to AFSMs in [15]; their difference is only in part (1) above. Wanda will try the framework first with dynamic DPs, and then with static DPs. If the initial DP problem given by the static approach is contained in the one for the dynamic approach, then the first step is skipped (so only the static approach is used).

6 Conclusions and future work

Overall, Wanda reads an input file, converts it into AFSM if needed, performs an analysis following §3–5 and then prints `YES` (the system terminates), `NO` (it does not terminate) or `MAYBE` (neither could be proved). In the first two cases, this is followed by a human-readable proof.

There are many directions for improvement. Most pertinently, Wanda is optimised for the format of the termination competition, and is weak when meta-variables take arguments. Also, non-termination analysis is very limited and does not take advantage of the DP framework. Other improvements could be to further extend first-order termination methods, build on primarily higher-order techniques like sized types [3], and weaken the pattern restriction in AFSMs.

A complete discussion of most techniques in Wanda and the technology behind automating them is available in the author's PhD thesis [15]. Wanda is open-source and available from:

http://wandahot.sourceforge.net/

References

[1] P. Aczel. A general Church-Rosser theorem. Unpublished Manuscript, University of Manchester. http://www.ens-lyon.fr/LIP/REWRITING/MISC/AGeneralChurch-RosserTheorem.pdf, 1978.

[2] F. Blanqui. HOT – an automated termination prover for higher-order rewriting. http://rewriting.gforge.inria.fr/hot.html.

[3] F. Blanqui. A type-based termination criterion for dependently-typed higher-order rewrite systems. In *Proc. RTA*, volume 3091 of *LNCS*, pages 24–39, 2004.

[4] F. Blanqui, J. Jouannaud, and M. Okada. Inductive-data-type systems. *Theoretical Computer Science*, 272(1-2):41–68, 2002.

[5] F. Blanqui, J. Jouannaud, and A. Rubio. The computability path ordering: The end of a quest. In *Proc. CSL*, volume 5213 of *LNCS*, pages 1–14, 2008.

[6] C. Borralleras and A. Rubio. THOR – an automatic tool for proving termination of higher-order rewriting. https://www.cs.upc.edu/~albert/term.html.

[7] Community. Confluence Competition (CoCo). http://coco.nue.riec.tohoku.ac.jp/.

[8] C. Fuhs and C. Kop. Harnessing first order termination provers using higher order dependency pairs. In *Proc. FroCoS*, volume 6989 of *LNAI*, pages 147–162, 2011.

[9] C. Fuhs and C. Kop. Polynomial interpretations for higher-order rewriting. In *Proc. RTA*, volume 15 of *LIPIcs*, pages 176–192, 2012.

[10] C. Fuhs and C. Kop. A static higher-order dependency pair framework. In *Proc. ESOP*, volume 11423 of *LNCS*, pages 752–782, 2019.

[11] J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *Proc. LPAR*, volume 3452 of *LNAI*, pages 301–331. Springer, 2005.

[12] M. Hamana. PolySOL – an automatic tool for confluence and termination of polymorphic second-order systems. http://www.cs.gunma-u.ac.jp/hamana/polysol/.

[13] J.W. Klop, V. van Oostrom, and F. van Raamsdonk. Combinatory reduction systems: introduction and survey. *Theoretical Computer Science*, 121(1-2):279 – 308, 1993.

[14] C. Kop. Simplifying algebraic functional systems. In *Proc. CAI*, volume 6742 of *LNCS*, pages 201–215. Springer, 2011.

[15] C. Kop. *Higher Order Termination*. PhD thesis, VU University Amsterdam, 2012.

[16] C. Kop and F. van Raamsdonk. Dynamic dependency pairs for algebraic functional systems. *Logical Methods in Computer Science*, 8(2):10:1–10:51, 2012. Special Issue for RTA '11.

[17] J. Nagele. CoCo 2016 participant: CSIˆho 0.2. http://coco.nue.riec.tohoku.ac.jp/2016/papers/csiho.pdf; tool webpage: http://cl-informatik.uibk.ac.at/software/csi/ho/.

[18] K. Onozawa, K. Kikuchi, T. Aoto, and Y. Toyama. ACPH: System description for CoCo 2016. http://coco.nue.riec.tohoku.ac.jp/2016/papers/acph.pdf.

[19] J.C. van de Pol. *Termination of Higher-order Rewrite Systems*. PhD thesis, University of Utrecht, 1996.

[20] M. Sakai, Y. Watanabe, and T. Sakabe. An extension of the dependency pair method for proving termination of higher-order rewrite systems. *IEICE Transactions on Information and Systems*, E84-D(8):1025–1032, 2001.

[21] S. Suzuki, K. Kusakari, and F. Blanqui. Argument filterings and usable rules in higher-order rewrite systems. *IPSJ Transactions on Programming*, 4(2):1–12, 2011.

[22] Wiki. Termination Competition. http://www.termination-portal.org/wiki/Termination_Competition.

On the Ground Confluence of Order-sorted Conditional Specifications Modulo Axioms: Maude's Church-Rosser Checker*

Francisco Durán[1], José Meseguer[2], and Camilo Rocha[3]

[1] University of Málaga, Málaga, Spain
[2] University of Illinois, Urbana-Champaign, IL, USA
[3] Pontificia Universidad Javeriana, Cali, Colombia

Extended Abstract

Checking local confluence by computing critical pairs is the standard way to check ground confluence. However, a perfectly reasonable equational program may be not locally confluent, and it can be very hard, or even impossible, to make it so by adding more equations. Indeed, Knuth-Bendix completion can often lead to an infinite loop and, even if it were to eventually succeed, can result in a highly bloated and hard to understand specification.

The Maude's Church-Rosser Checker (CRC) can be used to prove the ground local confluence of order-sorted and possibly conditional equational programs modulo associativity/commutativity/identity axioms. It supports three complementary methods. Method 1 ([DM10, DM12]) uses non-joinable critical pairs as completion hints to either achieve local confluence or reduce the number of critical pairs. Method 2 ([DMR18]) uses an inductive joinability inference system to try to prove the critical pairs ground joinable. Method 3 is used to prove the ground local confluence of a conditional equational specification whose conditions belong to a subspecification that has already been proved ground confluent and operationally terminating, and that is conservatively extended by the overall specification.

Method 1 follows the suggestion of Knuth-Bendix. Since failure of a proof of local confluence will generate a set of unjoinable critical pairs characterizing the most general cases in which rules cannot be shown confluent, such critical pairs can be used as useful hints for a user to modify his/her specification. If the new specification is locally confluent, operationally terminating, and sort-decreasing, we are done; otherwise, we can repeat the process on the new specification. This leads to an iterative and incremental process, in which the user repeatedly modifies his/her specification and re-checks the specification.

In practice, Method 1 works reasonably well, but it does not always lead to a locally confluent and sort-decreasing specification. Once we reach the limits of Method 1, that is, we cannot further modify or extend our specification without making it non-terminating, or simply too complex, we can try to prove the *ground joinability* of the remaining critical pairs using the inductive joinability inference system proposed in [DMR18]. Some of these pending critical pairs may be conditional. Although the inference system do not directly handles the conditional case, it can still be used in a conditional setting for a certain type of critical pairs.

Even though one could try to inductively prove the ground joinability of the original critical pairs, by first applying Method 1 we may significantly reduce the number of critical pairs, therefore reducing the proving effort. Furthermore, the user may have made mistakes in the original specification, so that the rest of the process becomes meaningless. Our experience shows that Method 1 has, as a side effect, the capability of revealing user mistakes, simply because it helps us to understand and identify

*Acknowledgements. Work partly funded by the project *PGC2018-094905-B-I00* (Spanish MINECO/FEDER), and by Univ. Málaga, Andalucía Tech.

potential issues. In such a case, Method 1 can be quite helpful in identifying such mistakes and help the user *restart* the process with a new specification. However, by adding new equations we may have changed the initial algebra semantics of the original specification. To ensure preservation of the initial algebra semantics and the ground confluence of the original specification, we can use the same inductive methods to prove ground joinability of all the equations added along the first step.

We still have one problem left. For operationally terminating conditional specifications it is in general *undecidable* whether they are confluent. We may, for example, have a conditional critical pair whose condition is unsatisfiable and therefore causes no confluence problems; but proving such unsatisfiability may be undecidable. One way to go would be to extend the inductive inference system to prove joinability of conditional critical pairs. We however are currently working on a method that seems to be more promising. The idea is rather simple, and basically consists in exploiting the hierarchical nature of most specifications: we assume an operationally terminating specification $(\Sigma, E \cup B)$, which has a subspecification $(\Sigma_0, E_0 \cup B_0)$ that has already been shown to be ground convergent and, furthermore: (i) the conditions in all equations in $E \setminus E_0$ are Σ_0-conditions and remain so after applying any substitutions; and (ii) $(\Sigma, E \cup B)$ conservatively extends $(\Sigma_0, E_0 \cup B_0)$ in the rewriting sense of not introducing any new rewrites among Σ_0-terms. Then, we can use the fact that $(\Sigma_0, E_0 \cup B_0)$ is convergent and apply the Church-Rosser Theorem to reason about the satisfiability/unsatisfiability of conditions in conditional critical pairs at the inductive equational logic level.

In practice, the opportunities for applying Method 3 to order-sorted conditional specifications are quite common, because specifications are typically developed in a modular way and also because it is quite common that the conditions in conditional equations only involve a subset of functions that often belong to already-defined submodules.

References

[DM10] Francisco Durán and José Meseguer. A church-rosser checker tool for conditional order-sorted equational maude specifications. In Peter Csaba Ölveczky, editor, *8th International Workshop Rewriting Logic and Its Applications, WRLA*, volume 6381 of *Lecture Notes in Computer Science*, pages 69–85. Springer, 2010.

[DM12] Francisco Durán and José Meseguer. On the Church-Rosser and coherence properties of conditional order-sorted rewrite theories. *Journal of Logic and Algebraic Programming*, 81(7-8):816–850, 2012.

[DMR18] Francisco Durán, José Meseguer, and Camilo Rocha. Proving ground confluence of equational specifications modulo axioms. In Vlad Rusu, editor, *12th International Workshop on Rewriting Logic and Its Applications, WRLA*, volume 11152 of *Lecture Notes in Computer Science*, pages 184–204. Springer, 2018.

Proving Non-Joinability using Weakly Monotone Algebras

Bertram Felgenhauer[1] and Johannes Waldmann[2]

[1] AoE, int-e@gmx.de
[2] HTWK Leipzig, johannes.waldmann@htwk-leipzig.de

Abstract

We propose a new non-joinability criterion based on weakly monotone algebras. It generalizes the tree automata based approach of over-approximating the sets of successors of each of the two terms. In contrast to Aoto's criterion based on weakly monotone algebras and usable rules, our approach uses different interpretations for the two source terms. The criterion has been implemented for string rewriting in noko-leipzig, a participant of the 2019 Confluence Competition. The algebras are given by arctically weighted finite automata, found by constraint solving.

1 Introduction

A TRS $\mathcal{R}$ is non-confluent if there are terms s, t that are convertible, but not joinable, that is, $\to_{\mathcal{R}}^*(s) \cap \to_{\mathcal{R}}^*(t) = \varnothing$. So, a semi-algorithm for proving non-confluence is to enumerate convertible s and t, and look for certificates of their non-joinability.

If $\mathcal{R}$ is non-terminating, then $\to_{\mathcal{R}}^*(s)$ or $\to_{\mathcal{R}}^*(t)$ can be infinite. In order to establish non-joinability, these sets, or some over-approximations, need to be represented in some finite way. Such a representation can also be useful if these sets are finite, but impractically large.

Zankl et al. [8] represent sets of reachable terms by finite tree automata. We employ weighted automata.

For classical automata A, B over-approximating $\to_{\mathcal{R}}^*(s)$ and $\to_{\mathcal{R}}^*(t)$, respectively, emptiness of $\mathcal{L}(A) \cap \mathcal{L}(B)$ is non-accessibility of the final states in the Cartesian product automaton $A \times B$. For weighted automata, we instead check that their Kronecker product algebra has bounded weights, and relate the bound to the Kronecker product $A(s) \cdot B(t)$.

This improves on classical automata in that we can handle some non-regular sets. In particular, arctically weighted automata can count and compare numbers of occurrences of symbols.

Example 1. We establish non-joinability of ag and bh with respect to the rules

$$g \to ag \qquad g \to i \qquad h \to bh \qquad h \to i \qquad i \to abi \qquad ab \to ba \qquad ba \to ab.$$

Using arctically weighted automata one can show that $\#_a(s) - \#_b(s) \geq 1$ ($\#_a(s)$ denotes the number of occurrences of a in s) for strings s reachable from ag, $\#_b(s) - \#_a(s) \geq 1$ for strings s reachable from bh, and that the sum of these two interpretations is 0 for all strings, which is smaller than $1 + 1 = 2$. Separating the successor of ag and bh using regular languages fails.

Our implementation of this method in noko-leipzig currently only handles the string rewriting case (one symbol is nullary, all others are unary). Noko-leipzig took part in the 2019 Confluence Competition,[1] where it achieved the highest number of NO answers in the SRS category, namely 34. The runner-up CSI had 28 NO answers.

[1] CoCo 2019, http://project-coco.uibk.ac.at/2019/

2 Preliminaries

We assume familiarity with term rewriting [2]. In the following, Σ denotes a signature, $\mathcal{X}$ a set of variables, $\mathcal{R}$ a term or string rewrite system (TRS or SRS), $\to_{\mathcal{R}}$ a single rewrite step, $\to_{\mathcal{R}}^*$ reachability, and s, t, u denote terms (we regard strings as terms).

Given a signature Σ, a Σ-algebra $\mathcal{A}$ has a carrier A and, for each function symbol $f \in \Sigma$ with arity n, an interpretation $f^{\mathcal{A}} : A^n \to A$. Given an assignment $\alpha : \mathcal{X} \to A$, the interpretation $[t]_\alpha^{\mathcal{A}}$ of a term s is defined inductively by

$$[t]_\alpha^{\mathcal{A}} = \begin{cases} [f(t_1, \ldots, t_n)]_\alpha^{\mathcal{A}} = f^{\mathcal{A}}([t_1]_\alpha^{\mathcal{A}}, \ldots, [t_n]_\alpha^{\mathcal{A}}) & \text{if } t = f(t_1, \ldots, t_n) \text{ for } f \in \Sigma \\ \alpha(t) & \text{if } t \in \mathcal{X}. \end{cases}$$

For ground terms t we may omit the α, writing $[t]^{\mathcal{A}}$. A weakly monotone Σ-algebra $\mathcal{A}$ is a Σ-algebra equipped with a partial order $\le$ on the carrier that satisfies $f^{\mathcal{A}}(a_1, \ldots, a_n) \le f^{\mathcal{A}}(a'_1, \ldots, a'_n)$ whenever $a_i \le a'_i$ for all $1 \le i \le n$. Let $\mathcal{A}$ and $\mathcal{B}$ be weakly monotone Σ-algebras. A map $h : A \to B$ is a pre-homomorphism if it is weakly monotone ($a \le a'$ implies $h(a) \le h(a')$) and satisfies $h(f^{\mathcal{A}}(a_1, \ldots, a_n)) \le f^{\mathcal{B}}(h(a_1), \ldots, h(a_n))$ for all $f \in \Sigma$ and $a_1, \ldots, a_n \in \mathcal{A}$.

A weakly monotone Σ-algebra $\mathcal{A}$ weakly orients a TRS $\mathcal{R}$ if we have $[l]_\alpha^{\mathcal{A}} \le [r]_\alpha^{\mathcal{A}}$ for all rules $l \to r \in \mathcal{R}$ and assignments $\alpha : \mathcal{X} \to A$. This implies $[s]_\alpha^{\mathcal{A}} \le [t]_\alpha^{\mathcal{A}}$ whenever $s \to_{\mathcal{R}}^* t$. Note that the order is flipped compared the convention for termination of TRSs.

3 Results

We start out by giving an abstract non-joinability criterion based on weakly monotone algebras.

Lemma 2. *Let Σ be a signature, $\mathcal{R}$ be a TRS over Σ and $s, t \in \mathcal{T}(\Sigma)$ ground terms. Furthermore let $\mathcal{A}$, $\mathcal{B}$ be weakly monotone Σ-algebras such that $\mathcal{R}$ is weakly oriented by both $\mathcal{A}$ and $\mathcal{B}$, and let $\mathcal{C}$ be a partially ordered set. Finally, let $\delta : \mathcal{A} \times \mathcal{B} \to \mathcal{C}$ be a weakly (bi-)monotone function. Then s and t are non-joinable if, for some $c \in \mathcal{C}$,*

1. *$\delta([s]^{\mathcal{A}}, [t]^{\mathcal{B}}) \not\le c$, and*
2. *$\delta([u]^{\mathcal{A}}, [u]^{\mathcal{B}}) \le c$ for all $u \in \mathcal{T}(\Sigma)$.*

Proof. Assume that s and t have a common reduct u: $s \to_{\mathcal{R}}^* u \,{}_{\mathcal{R}}^*\!\leftarrow t$. Then $[s]^{\mathcal{A}} \le [u]^{\mathcal{A}}$ and $[t]^{\mathcal{B}} \le [u]^{\mathcal{B}}$, because $\mathcal{R}$ is weakly oriented by $\mathcal{A}$ and $\mathcal{B}$. Consequently, by monotonicity of δ we have $\delta([s]^{\mathcal{A}}, [t]^{\mathcal{B}}) \le \delta([u]^{\mathcal{A}}, [u]^{\mathcal{B}}) \le c$, contradicting the first assumption. □

In order to obtain a finite criterion, we extend $\mathcal{C}$ to be a weakly monotone algebra as well, leading to the following result.

Theorem 3. *Let $\mathcal{A}$, $\mathcal{B}$, $\mathcal{C}$ be weakly monotone Σ-algebras such that $\mathcal{R}$ is weakly oriented by both $\mathcal{A}$ and $\mathcal{B}$, $s, t \in \mathcal{T}(\Sigma)$ be ground terms and $\delta : \mathcal{A} \times \mathcal{B} \to \mathcal{C}$ be a pre-homomorphism between weakly monotone Σ-algebras. Then s and t are non-joinable provided that for some $c \in \mathcal{C}$,*

1. *$\delta([s]^{\mathcal{A}}, [t]^{\mathcal{B}}) \not\le c$, and*
2. *$f^{\mathcal{C}}(c, \ldots, c) \le c$ for all $f \in \Sigma$.*

Remark 4. The condition that δ is a pre-homomorphism between weakly monotone Σ-algebras means that δ is weakly monotone and that for all $f \in \Sigma$, $a_1, \ldots, a_n \in \mathcal{A}$, $b_1, \ldots, b_n \in \mathcal{B}$,

$$\delta(f^{\mathcal{A}}(a_1, \ldots, a_n), f^{\mathcal{B}}(b_1, \ldots, b_n)) \le f^{\mathcal{C}}(\delta(a_1, b_1), \ldots, \delta(a_n, b_n)).$$

Proof. First we show that $\delta([u]^{\mathcal{A}}, [u]^{\mathcal{B}}) \leq c$ for all ground terms u by induction on u:

$$\begin{aligned}\delta([f(u_1,\ldots,u_n)]^{\mathcal{A}}, [f(u_1,\ldots,u_n)]^{\mathcal{B}}) &= \delta(f^{\mathcal{A}}([u_1]^{\mathcal{A}},\ldots,[u_n]^{\mathcal{A}}), f^{\mathcal{B}}([u_1]^{\mathcal{B}},\ldots,[u_n]^{\mathcal{B}}))\\ &\leq f^{\mathcal{C}}(\delta([u_1]^{\mathcal{A}},[u_1]^{\mathcal{B}}),\ldots,\delta([u_n]^{\mathcal{A}},[u_n]^{\mathcal{B}}))\\ &\leq f^{\mathcal{C}}(c,\ldots,c) \leq c.\end{aligned}$$

We conclude by Lemma 2. □

4 Algebras from Finite Weighted Automata

We instantiate the general approach to algebras represented by finite weighted automata over some ordered semi-ring [5] $(S, 0, 1, +, \cdot, \leq)$ where multiplication is commutative, the order is weakly monotonic w.r.t. both operations, and S is positive ($\forall s \in S : 0 \leq s$).

Examples of such semi-rings are the natural numbers $\mathbb{N}$ with standard addition, product, and order; Booleans $(\mathbb{B}, \mathbf{F}, \mathbf{T}, \vee, \wedge, \leq)$ with $\mathbf{F} < \mathbf{T}$; and arctic integers $(\mathbb{A}, -\infty, 0, \max, +, \leq)$ where $\mathbb{A} = \{-\infty\} \cup \mathbb{Z}$, and operations on $\mathbb{Z}$ extended suitably.

An S-weighted tree automaton [3] A over alphabet Σ, with set of states Q, is given by a family of transition mappings $\mu_k : \Sigma_k \to (Q^k \times Q \to S)$, and a root weight vector $\nu : Q \to S$. The algebra μ_A of this automaton has domain $(Q \to S, \leq)$. These are Q-indexed vectors of S values, ordered point-wise. Operations of μ_A are defined via μ_k in a standard way.

We note the special case of weighted word automata where $\mu_0(\epsilon)$ is a Q-vector, and $\mu_1(f)$ is a Q-by-Q matrix that operates on the domain by multiplication from the right. (To make the connection to terms, we regard the string fg as $\epsilon fg = g(f(\epsilon))$.)

Under the given restrictions, the cross product automaton $A \times B$ computes the Hadamard product $\mu_A \odot \mu_B$, which is a pre-homomorphism of algebras.

Example 5. We revisit the SRS from the introduction consisting of the rules

$$g \to ag \qquad g \to i \qquad h \to bh \qquad h \to i \qquad i \to abi \qquad ab \to ba \qquad ba \to ab.$$

We use the following interpretations over the arctic integers for $\mathcal{A}$ and $\mathcal{B}$:

$$\begin{array}{lllllll}\epsilon^{\mathcal{A}} = 0 & a^{\mathcal{A}}(x) = x+1 & b^{\mathcal{A}}(x) = x-1 & g^{\mathcal{A}}(x) = x & h^{\mathcal{A}}(x) = -\infty & i^{\mathcal{A}}(x) = x\\ \epsilon^{\mathcal{B}} = 0 & a^{\mathcal{B}}(x) = x-1 & b^{\mathcal{B}}(x) = x+1 & g^{\mathcal{B}}(x) = -\infty & h^{\mathcal{B}}(x) = x & i^{\mathcal{B}}(x) = x\end{array}$$

This corresponds to arctic automata with a single state that are compatible with all the given rules. In this case the Kronecker product is just addition, and it's easy to verify that $x \leq 0$ implies $a^{\mathcal{A}}(x) + a^{\mathcal{B}}(x) \leq 0$, $b^{\mathcal{A}}(x) + b^{\mathcal{B}}(x) \leq 0$, and so on. On the other hand, $[ag]^{\mathcal{A}} + [bh]^{\mathcal{B}} = 1 + 1 \not\leq 0$, so ag and bh are not joinable.

Note that due to the rules $ab \to ba$, $ba \to ab$, and $i \to abi$ we have joinability of s and t whenever $s, t \in \{a, b\}^* i$ and $\#_a(s) - \#_b(s) = \#_a(t) - \#_b(t)$. Since we can use g (h) to produce an arbitrary number of extra a-s (b-s) starting from ag (bh) before producing an i, counting the number of a-s and b-s is essential for proving non-joinability of ag and bh. This is impossible with finitely many states, so the classical automata approach fails.

5 Implementation: Noko Leipzig

Noko-leipzig is a spin-off of the Matchbox termination prover [7]. It contains the following semi-algorithm for disproving joinability:

- input: SRS $\mathcal{R}$ over Σ, words $s, t \in \Sigma^*$, numbers $d, b \in \mathbb{N}$.
- output (if successful): arctically weighted automata A, B, and arctic vector $c \in (Q_A \times Q_B \to \mathbb{A})$, such that the algebras $\mu_A, \mu_B, \mu_A \odot \mu_B$ and vector c fulfil the conditions of Theorem 3, where A and B have d states and all arctic numbers are represented by b bits.

The implementation works via transformation to a Boolean satisfiability problem, using the Ersatz library [6]. We briefly analyze noko-leipzig's six unique NO answers in the SRS category of CoCo 2019. Only two of these answers (Cops[2] 1034 and 1131) use the technique described here. In the case of Cop 1034, the languages $\to^*_{\mathcal{R}}(s)$ and $\to^*_{\mathcal{R}}(t)$ are finite, so let us focus on Cop 1131.

Example 6. Cop 1131 is the SRS $\mathcal{R} = \{a \to caa, b \to aca, b \to acb, b \to bab, c \to cac\}$. Noko-leipzig proves that $s = aca \leftarrow \cdot \to bab = t$ is non-joinable, using two-state automata. With different search parameters (fixing the number of states to one), we obtain a shorter proof that is similar to the one in Example 1: A is a one-state automaton that computes the function

$$A : w \mapsto \text{if } b \in w \text{ then } -\infty \text{ else } \#_c(w) - \#_a(w),$$

(this is accomplished by assigning weights -1, $-\infty$, 1 and 0 to a, b, c and ϵ, respectively) and B is a one-state automaton that computes the function

$$B : w \mapsto \#_a(w) + \#_b(w) - \#_c(w).$$

The product automaton $A \times B$ computes the function

$$C : w \mapsto \text{if } b \in w \text{ then } -\infty \text{ else } 0.$$

We have $A(s) = -1$ and $B(t) = 3$, thus, $A(s) \cdot B(t) = 2 \not\leq C(w)$.

6 Related Work

The following example shows that our criterion generalizes non-joinability analysis using compatible tree automata. [4, 8]

Example 7 (compatible tree automata). Recall the following criterion: Let $A = (Q_A, F_A, \Delta_A)$, $B = (Q_B, F_B, \Delta_B)$ be tree automata that are compatible with $\mathcal{R}$. If $s \in \mathcal{L}(A)$, $t \in \mathcal{L}(B)$, and $\mathcal{L}(A) \cap \mathcal{L}(B) = \varnothing$, then s and t are not joinable. We show that this criterion is an instance of Lemma 2 and of Theorem 3.

The automaton A gives rise to a weakly monotone Σ-algebra $\mathcal{A}$ on 2^{Q_A} with $\subseteq$ as the underlying order and interpretations for $f \in \Sigma$ given by the transition functions

$$f^{\mathcal{A}}(Q_1, \ldots, Q_n) = \{q \mid f(q_1, \ldots, q_n) \to q \in \Delta_A \text{ and } q_i \in Q_i \text{ for } 1 \leq i \leq n\}.$$

Compatibility of A with $\mathcal{R}$ implies that $\mathcal{A}$ weakly orients $\mathcal{R}$. The same considerations apply to the automaton B, giving rise to a weakly monotone Σ-algebra $\mathcal{B}$ that weakly orients $\mathcal{R}$.

Using these algebras, the memberships $s \in \mathcal{L}(A)$, $t \in \mathcal{L}(B)$ translate to $[s]^{\mathcal{A}} \not\subseteq Q_A - F_A$ and $[t]^{\mathcal{B}} \not\subseteq Q_B - F_B$, respectively, which is equivalent to $[s]^{\mathcal{A}} \times [t]^{\mathcal{B}} \not\subseteq Q_A \times Q_B - F_A \times F_B$. The statement $\mathcal{L}(A) \cap \mathcal{L}(B) = \varnothing$ can be expressed as $[u]^{\mathcal{A}} \times [u]^{\mathcal{B}} \subseteq Q_A \times Q_B - F_A \times F_B$ for all ground terms u. By Lemma 2, non-joinability of s and t follows, using $\mathcal{C} = 2^{Q_A \times Q_B}$ with $\subseteq$ as the underlying order, $\delta(P, Q) = P \times Q$, and $c = Q_A \times Q_B - F_A \times F_B$.

[2] See the **co**nfluence **p**roblem**s** database, https://cops.uibk.ac.at/

To make Theorem 3 applicable, consider the intersection product tree automaton $A \times B$ with final states $F_A \times F_B$. Let $\mathcal{C}$ be the corresponding weakly monotone Σ-algebra on $2^{Q_A \times Q_B}$, ordered by $\subseteq$. The set R of reachable states of $A \times B$ satisfies $f^{\mathcal{C}}(R, \ldots, R) \subseteq R$ for all $f \in \Sigma$; in fact it is the smallest set with this property. Furthermore, since $R \subseteq Q_A \times Q_B - F_A \times F_B$ (otherwise $\mathcal{L}(A) \cap \mathcal{L}(B)$ would be non-empty), we have $[s]^{\mathcal{A}} \times [t]^{\mathcal{B}} \not\subseteq R$. So Theorem 3 applies using $\delta(P, Q) = P \times Q$ and $c = R$.

In fact, a classical non-weighted tree automaton is a $\mathbb{B}$-weighted tree automaton, and the non-weighted intersection product automaton is the Hadamard product.

We believe that our results are incomparable to Aoto's criterion based on weakly monotone algebras and usable rules [1]. Let us recall the result:

Theorem 8 (Aoto 2013)**.** *Let $\mathcal{D}$ be a weakly monotone Σ-algebra such that $\mathcal{U}(s, \mathcal{R})$ is weakly oriented by $(\mathcal{D}, \leq)$ and $\mathcal{U}(t, \mathcal{R})$ is weakly oriented by $(\mathcal{D}, \geq)$. Then s, t are non-joinable provided that $[s]^{\mathcal{D}} \not\leq [t]^{\mathcal{D}}$.*

One difference to our results is the incorporation of usable rules, denoted by $\mathcal{U}(s, \mathcal{R})$. Ideally, this would be the set of rules that are applicable to successors of s; in practice, it is an over-approximation of that set. We leave the discussion of usable rules for future work.

The other notable difference is that only one interpretation $\mathcal{D}$ is used. We can fit that into the setup of Lemma 2 by using $(\mathcal{D}, \leq)$ for $\mathcal{A}$, $(\mathcal{D}, \geq)$ for $\mathcal{B}$, $\mathcal{A} \times \mathcal{B}$ for $\mathcal{C}$, and $\delta(a, b) = (a, b)$. Then $s \to_{\mathcal{R}}^* u \,{}_{\mathcal{R}}^*\!\leftarrow t$ would imply $\delta([s]^{\mathcal{A}}, [t]^{\mathcal{B}}) \leq \delta([u]^{\mathcal{A}}, [u]^{\mathcal{B}})$. The latter is equivalent to $[s]^{\mathcal{D}} \leq [u]^{\mathcal{D}} \leq [t]^{\mathcal{D}}$, contradicting the assumption $[s]^{\mathcal{D}} \not\leq [t]^{\mathcal{D}}$, so non-joinability of s and t follows. Note that the term u is not analyzed at all; the only fact about u that is used in this argument is that $[u]^{\mathcal{A}} = [u]^{\mathcal{D}} = [u]^{\mathcal{B}}$. This is in stark contrast to the idea underlying Theorem 3, namely to establish an upper bound for $[u]^{\mathcal{C}}$ for all ground terms u.

References

[1] T. Aoto. Disproving confluence of term rewriting systems by interpretation and ordering. In *Proc. 9th FroCoS*, volume 8152 of *LNCS (LNAI)*, pages 311–326, 2013.

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

[3] Z. Fülöp and H. Vogler. Weighted tree automata and tree transducers. In M. Droste, W. Kuich, and H. Vogler, editors, *Handbook of Weighted Automata*, chapter 9, pages 313–403. Springer, 2009.

[4] T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In *Proc. 9th RTA*, volume 1379 of *LNCS*, pages 151–165, 1998.

[5] J.S. Golan. *Semirings and Affine Equations over Them.* Kluwer, 2003.

[6] E. Kmett, E. Mertens, and J. Kiviniemi. ersatz: A monad for expressing SAT or QSAT problems using observable sharing. https://hackage.haskell.org/package/ersatz, 2018.

[7] J. Waldmann. The matchbox termination prover. https://gitlab.imn.htwk-leipzig.de/waldmann/pure-matchbox, 2019.

[8] H. Zankl, B. Felgenhauer, and A. Middeldorp. CSI – A confluence tool. In *Proc. 23rd CADE*, volume 6803 of *LNCS (LNAI)*, pages 499–505, 2011.

The Diamond Lemma for non-terminating rewriting systems using deterministic reduction strategies

Cyrille Chenavier[1] and Maxime Lucas[2]

[1] Inria Lille - Nord Europe, Équipe Valse
cyrille.chenavier@inria.fr
[2] Inria Rennes - Bretagne Atlantique, Équipe Gallinette
maxime.lucas@inria.fr

Abstract

We study the confluence property for rewriting systems whose underlying set of terms admits a vector space structure. For that, we use deterministic reduction strategies. These strategies are based on the choice of standard reductions applied to basis elements. We provide a sufficient condition of confluence in terms of the kernel of the operator which computes standard normal forms. We present a local criterion which enables us to check the confluence property in this framework. We show how this criterion is related to the Diamond Lemma for terminating rewriting systems.

1 Introduction

The fact that local confluence together with termination implies confluence has been known for abstract rewriting systems since Newman's work [8]. For rewriting on noncommutative polynomials, a similar result known as the Diamond lemma was introduced by Bergman [2] nearly 30 years later, in order to compute normal forms in noncommutative algebras using rewriting theory. It asserts that for terminating rewriting systems, the local confluence property can be checked on monomials.

One difficulty of rewriting polynomials is that the naive notion of rewriting path (obtained as the closure of the generating rewriting relations under reflexivity, transitivity, sum and product by a scalar) does not terminate. Instead, one needs to first consider well-formed rewriting steps before forming the reflexive transitive closure.

Nevertheless the Diamond lemma has proved to be very useful : together with the works of Bokut [3], it has given birth the theory of noncommutative Gröbner bases [7]. The latter have provided applications to various areas of noncommutative algebra such as the study of embedding problems (which appear in the works of Bokut and Bergman), homological algebra [4, 5] or Koszul duality [1, 9].

Computation of normal forms in noncommutative algebra is also used to provide formal solutions to partial differential equations. In this framework, a confluence criterion analogous to the Diamond Lemma is given by Janet bases [10], which specify a deterministic way to reduce each polynomial into normal form using standard reductions [6]. The confluence criterion may then be asserted as follows: for each monomial m and each non-standard reduction $m \to f$, f is reducible into $\widehat{m}$, where the latter is obtained from m using only standard reductions.

In the presented paper, we propose an extension of the Diamond Lemma which offers two improvements over the one from Bergman: first it allows the treatment of non-terminating rewriting relations, and second it does not presuppose a notion of well-formed rewriting steps. This last property seems particularly promising in order to extend the Diamond Lemma to other structures.

Instead of supposing that the rewriting relation studied is terminating, we suppose given an ordering on the monomials, independent of the rewriting relation. We then use methods based on standard reductions: for every monomial m, we select exactly one reduction with left-hand side m, which is decreasing for the ordering chosen. Such choices induce a deterministic way to reduce each polynomial, obtained by applying simultaneously standard reductions on every monomial appearing in its decomposition. When these deterministic reductions terminate, one defines an operator which maps every polynomial to its unique standard normal form.

From this operator, we define a suitable notion of confluence in our setting, and show in Proposition 3.4 that it implies the usual notion of confluence for the rewriting system studied. We then provide an effective method for checking this criterion in Theorem 3.7. This method is based on a local analysis corresponding to checking local confluence on monomials. In particular, when the rewriting system is terminating, we show (Theorem 3.9) that we recover the Diamond Lemma as a particular case of Theorem 3.7.

2 Local strategies and h-normal forms

We fix a commutative field $\mathbb{K}$ as well as a well-founded partially ordered set $(X, <_X)$. We denote by $\mathbb{K}X$ the vector space spanned by X: an element $v \in \mathbb{K}X$ is a finite formal linear combination of elements of X with coefficients in $\mathbb{K}$. The sum of $u = \sum \lambda_x x$ and $v = \sum \mu_x x$ equals $\sum(\lambda_x + \mu_x)x$ and the product of $\lambda \in \mathbb{K}$ by v equals $\sum(\lambda\lambda_x)x$. For every $v \in \mathbb{K}X$, there exists a unique finite set $\mathrm{supp}(v) \subset X$, called the *support* of v, such that

$$v = \sum_{x \in \mathrm{supp}(v)} \lambda_x x \text{ and } x \in \mathrm{supp}(v) \Rightarrow \lambda_x \neq 0. \tag{1}$$

We extend the order $<_X$ into the multiset order on $\mathbb{K}X$, denoted $<_{\mathbb{K}X}$: for any $u, v \in \mathbb{K}X$, $u <_{\mathbb{K}X} v$ if $\mathrm{supp}(u) \neq \mathrm{supp}(v)$ and for any $x \in \mathrm{supp}(u) \setminus \mathrm{supp}(v)$, there exists $y \in \mathrm{supp}(v) \setminus \mathrm{supp}(u)$ such that $y >_X x$. Note that $<_{\mathbb{K}X}$ and $<_X$ coincide when restricted to X, so we simply denote this order by $<$ in the rest of this paper.

We fix a set $R \subseteq X \times \mathbb{K}X$ which represents rewrite rules of the form $x \underset{R}{\longrightarrow} r$. The set R induces a rewriting relation on $\mathbb{K}X$ which reduces many x's at once and defined as follows:

$$\sum_x \lambda_x x + v \underset{R}{\longrightarrow} \sum_x \lambda_x r_x + v, \tag{2}$$

where v is any element of $\mathbb{K}X$, and for any $x \in X$ appearing in the sum, $\lambda_x \neq 0$, $x \underset{R}{\longrightarrow} r_x \in R$ and $x \notin \mathrm{supp}(v)$. Finally we denote by $\overset{*}{\underset{R}{\longleftrightarrow}}$ the closure of $\underset{R}{\longrightarrow}$ under transitivity, symmetry and sum.

Definition 2.1. A *local strategy* h for R is the choice, for every $x \in X$ not minimal for $<$, of a rewriting rule $h_x = x \underset{R}{\longrightarrow} r_x$ such that $r_x < x$.

In the rest of this paper, we suppose chosen such a local strategy h (note that such an h may not exist). Any vector v can be decomposed in a unique way as $\sum \lambda_x x + v'$, where $y \in \mathrm{supp}(v')$ implies that y is minimal for $<$, and $x \in \mathrm{supp}(v) \setminus \mathrm{supp}(v')$ is not. We define a rewriting relation $\underset{h}{\longrightarrow}$ as follows:

$$h_v = \sum \lambda_x x + v' \underset{h}{\longrightarrow} \sum \lambda_x r_x + v', \tag{3}$$

where for every x, $h_x = x \underset{R}{\longrightarrow} r_x$. Note in particular that if x is minimal for $<$, then $h_x = x \to x$ is the identity on x.

Definition 2.2. A vector v is said to be an *h-normal form* if it is a normal form for $\underset{h}{\longrightarrow}$.

Example 2.3. Let $X = \{x, y, z, t\}$, $x \underset{R}{\longrightarrow} y$, $y \underset{R}{\longrightarrow} z + t$, $z \underset{R}{\longrightarrow} y - t$. Note that this is not terminating since we have the infinite loop $y \underset{R}{\longrightarrow} z + t \underset{R}{\longrightarrow} (y - t) + t = y$. We choose the order $x > y > z, t$, and the following distinguished rewrite rules: $h_x = x \underset{h}{\longrightarrow} y$ and $h_y = y \underset{h}{\longrightarrow} z + t$. Then the R-normal forms are the $\lambda_t t$, while the h-normal forms are all the expressions of the form $\lambda_t t + \lambda_z z$.

Lemma 2.4. *Let v be a vector in $\mathbb{K}X$. Either v is minimal for $<$, or there exists $v' < v$ such that $v \underset{h}{\longrightarrow} v'$. In particular, h-normal forms are precisely the minimal elements of $\mathbb{K}X$ for $<$.*

For each $v \in \mathbb{K}X$ and strategy local strategy h, there exists at most one v' such that $v \underset{h}{\longrightarrow} v'$, and $\underset{h}{\longrightarrow}$ is compatible with the termination order $<$. As a consequence, any $v \in \mathbb{K}X$ is sent by multiple applications of $\underset{h}{\longrightarrow}$ to a unique h-normal form that we denote by $H(v)$. This defines a map $H : \mathbb{K}X \to \mathbb{K}X$.

Proposition 2.5. *The map H is a linear projector, in the sense that for all $u, v \in \mathbb{K}X$ and $\lambda \in \mathbb{K}$, $H(u + v) = H(u) + H(v)$, $H(\lambda u) = \lambda H(u)$ and $H(H(u)) = H(u)$.*

Proof. The h-normal forms are closed under sums, so that $H(H(v)) = H(v)$ for every v, that is H is a projector. Moreover, if $u \underset{h}{\longrightarrow} u'$ and $v \underset{h}{\longrightarrow} v'$, then we have $u + v \underset{h}{\longrightarrow} u' + v'$. Iterating $\underset{h}{\longrightarrow}$, we get $H(u + v) = H(H(u) + H(v)) = H(u) + H(v)$. □

3 A confluence criterion

In this section we investigate the confluence properties of R. The main idea behind this section is that under suitable hypothesis $\underset{h}{\longrightarrow}$ should form a terminating, confluent subrelation of $\underset{R}{\longrightarrow}$.

We start in Definition 3.1 and the following propositions by relating the confluence of $\underset{R}{\longrightarrow}$ to properties on h. Then Theorem 3.7, we prove a confluence criterion to check whether R satisfies Definition 3.1.

Definition 3.1. We say that R is *h-confluent* if for every rewrite rule $x \underset{R}{\longrightarrow} v \in R$, we have $H(x - v) = 0$.

Example 3.2. Let us take the same example as in Example 2.3. We have three equations to check:

$$H(x) = z + t = H(y), \qquad H(y) = z + t = H(z + t), \qquad H(z) = z = H(y - t),$$

and so R is h-confluent. Replacing the rule $z \underset{R}{\longrightarrow} y - t$ by $z \underset{R}{\longrightarrow} y$, we get $H(z) = z$ and $H(y) = z + t$, so R is not h-confluent anymore.

Proposition 3.3. *If R is h-confluent, then $u \overset{*}{\underset{R}{\longleftrightarrow}} v$ if and only if $H(u - v) = 0$.*

Proof. The relation $\overset{*}{\underset{R}{\longleftrightarrow}}$ is the closure of $\underset{R}{\longrightarrow}$ under transitivity, symmetry and sum. Since the relation $H(u - v) = 0$ is closed under these operations, we get one implication.

Reciprocally, if $H(u - v) = 0$ then by definition of H we have $u \overset{*}{\underset{h}{\longleftrightarrow}} v$, and in particular $u \overset{*}{\underset{R}{\longleftrightarrow}} v$. □

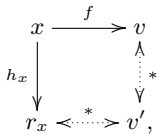
Proposition 3.4. *If R is h-confluent then $\underset{R}{\longrightarrow}$ is confluent.*

Proof. Let $v,\ v_1,\ v_2 \in \mathbb{K}X$ be such that $v \underset{R}{\overset{*}{\longrightarrow}} v_i$, for $i = 1,\ 2$. From Proposition 3.3, $v_1 - v_2$ belongs to $\ker(H)$, that is $H(v_1) = H(v_2)$. Denoting by u the common value, we get $v_i \underset{R}{\overset{*}{\longrightarrow}} u$, which proves the proposition. □

Note that the previous proposition is a sufficient but not a necessary condition: taking X to be the integers, with the relations $n \underset{R}{\longrightarrow} n+1$ is confluent, but there exist no local strategy h making R h-confluent.

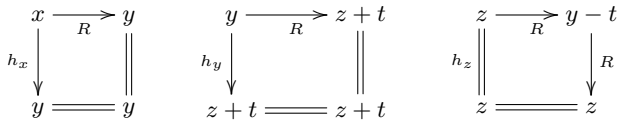
We now introduce our criterion to show that R is h-confluent. For that, we assume that the set of relations R is equipped with a well-founded order $\prec$ satisfying the following decreasingness property:

Definition 3.5. We say that R is *locally h-confluent* if for every $x \in X$ and $f = x \underset{R}{\longrightarrow} v$, then letting $h_x = x \underset{h}{\longrightarrow} r_x$, we have the confluence diagram:



where each rewriting step occurring in the dotted arrows is strictly smaller than f with respect to $\prec$.

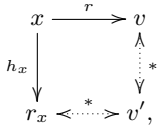
Example 3.6. Continuing with Example 2.3, let us define an order $\prec$ on R by the following ordering: $(x \underset{R}{\longrightarrow} y), (y \underset{R}{\longrightarrow} z+t) \prec (z \underset{R}{\longrightarrow} y-t)$. This is guided by the heuristic that rules advancing towards an h-normal form should be favored over rules that do not: here z is an h-normal form so the rule rewriting it is large for $\prec$. The following diagrams show that R is locally h-confluent:



Our main result is the following.

Theorem 3.7. *If R is locally h-confluent, then R is h-confluent. In particular, $\underset{R}{\longrightarrow}$ is confluent.*

Proof. We reason by induction on r according to the order $\prec$. Looking at the square corresponding to r:



we have $H(x) = H(r_x)$ by definition of H, and $H(r_x) = H(v') = H(v)$ by induction hypothesis, which concludes the proof. □

Remark 3.8. Local h-confluence implies that the pair of rewriting relations $(\underset{h}{\longrightarrow}, \underset{R}{\longrightarrow})$ is decreasing with respect to conversions (see [11, Definition 3]), using the order $\prec$ on R and the discrete ordering on $\underset{h}{\longrightarrow}$. By [11, Theorem 3], this implies that $(\underset{h}{\longrightarrow}, \underset{R}{\longrightarrow})$ commute. Using the fact that $\underset{h}{\longrightarrow} \subseteq \underset{R}{\longrightarrow}$, one can then recover that $\underset{R}{\longrightarrow}$ is confluent.

Let us show how the Diamond Lemma fits as a particular case of our setup.

Theorem 3.9 ([2]). *Assume that $\underset{R}{\longrightarrow}$ is terminating and that for every $x \in X$, $x \underset{R}{\longrightarrow} r$ and $x \underset{R}{\longrightarrow} r' \in R$, r and r' are joinable. Then, $\underset{R}{\longrightarrow}$ is confluent.*

Proof. We define an ordering $x > y$ on X as the transitive closure of the relation "there exists $v \in \mathbb{K}X$ such that $x \underset{R}{\longrightarrow} v$ and $y \in \mathrm{supp}(v)$". This is well-founded since by hypothesis $\underset{R}{\longrightarrow}$ is terminating. By definition, if $x \in X$ is not minimal for $>$, then x is not an R-normal form. Let us fix an arbitrary rewriting step $h_x = x \underset{h}{\longrightarrow} r_x$. By definition of $>$, for any $y \in \mathrm{supp}(r_x)$ we have $y < x$ and so $r_x < x$, which shows that h is a local strategy. Ordering the rewrite rules by their left hand sides makes R locally h-confluent. Theorem 3.7 finally shows that R is confluent. □

Conclusion. We introduced a sufficient condition, based on deterministic reduction strategies, of confluence for rewriting systems on vector spaces. As a particular case, we recover the Diamond Lemma. This work maybe extended in particular into two main directions. The first one consists in weakening our assumption on the set $\mathbb{K}$ of coefficients, by allowing non invertible coefficients. A second extension consists in characterising Janet bases in this framework, with the objective to develop constructive methods in the analysis and formal resolution of PDE's.

References

[1] Roland Berger. Koszulity for nonquadratic algebras. *J. Algebra*, 239(2):705–734, 2001.

[2] George M. Bergman. The diamond lemma for ring theory. *Adv. in Math.*, 29(2):178–218, 1978.

[3] Leonid A. Bokut′. Imbeddings into simple associative algebras. *Algebra i Logika*, 15(2):117–142, 245, 1976.

[4] Yuji Kobayashi. Complete rewriting systems and homology of monoid algebras. *J. Pure Appl. Algebra*, 65(3):263–275, 1990.

[5] Yuji Kobayashi. Gröbner bases of associative algebras and the Hochschild cohomology. *Trans. Amer. Math. Soc.*, 357(3):1095–1124, 2005.

[6] Paul-André Melliès. *Axiomatic Rewriting Theory I: A Diagrammatic Standardization Theorem*, pages 554–638. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[7] Teo Mora. An introduction to commutative and noncommutative Gröbner bases. *Theoret. Comput. Sci.*, 134(1):131–173, 1994.

[8] Maxwell H. A. Newman. On theories with a combinatorial definition of "equivalence.". *Ann. of Math. (2)*, 43:223–243, 1942.

[9] Stewart B. Priddy. Koszul resolutions. *Trans. Amer. Math. Soc.*, 152:39–60, 1970.

[10] Werner M. Seiler. Spencer cohomology, differential equations, and Pommaret bases. In *Gröbner bases in symbolic analysis*, volume 2 of *Radon Ser. Comput. Appl. Math.*, pages 169–216. Walter de Gruyter, Berlin, 2007.

[11] Vincent Van Oostrom. Confluence by decreasing diagrams. In *International Conference on Rewriting Techniques and Applications*, pages 306–320. Springer, 2008.

infChecker
A Tool for Checking Infeasibility*

Raúl Gutiérrez[1†] and Salvador Lucas[1]

Valencian Research Institute for Artificial Intelligence, Universitat Politècnica de València
Camino de Vera s/n, E-46022 Valencia, Spain
{rgutierrez,slucas}@dsic.upv.es

Abstract

Given a Conditional Term Rewriting System (CTRS) $\mathcal{R}$ and terms s and t, we say that the reachability condition $s \to^* t$ is *feasible* if there is a substitution σ instantiating the variables in s and t such that the *reachability test* $\sigma(s) \to^*_{\mathcal{R}} \sigma(t)$ succeeds; otherwise, we call it *infeasible*. Checking infeasibility of such (sequences of) reachability conditions is important in the analysis of computational properties of CTRSs, like confluence or operational termination. Recently, a logic-based approach to prove and disprove infeasibility has been introduced. In this paper we present infChecker, a new tool for checking infeasibility which is based on such an approach.

1 Introduction

When analyzing the computational behaviour of CTRSs $\mathcal{R}$, consisting of rules $\ell \to r \Leftarrow s_1 \approx t_1, \ldots, s_n \approx t_n$, we need to consider two kinds of computations: (1) the reduction of expressions in the usual way, i.e., by replacing an instance $\sigma(\ell)$ of the left-hand side ℓ by the instance $\sigma(r)$ of the right-hand side r using a matching substitution σ and (2) the evaluation of the conditions $s_i \approx t_i$ in the rules, which (for *oriented* CTRSs) are treated as reachability tests $\sigma(s_i) \to^* \sigma(t_i)$. In this setting, representing rewriting steps in CTRSs as proofs of goals in the logic of (oriented) CTRSs with inference system in Figure 1 becomes a natural way to represent computations [6]. Given a CTRS $\mathcal{R}$, an inference system $\mathcal{I}(\mathcal{R})$ is obtained from the inference rules in Figure 1 by *specializing* $(C)_{f,i}$ for each k-ary symbol f in the signature $\mathcal{F}$ and $1 \le i \le k$ and $(Rl)_\rho$ for all conditional rules $\rho : \ell \to r \Leftarrow c$ in $\mathcal{R}$. We write $s \to_{\mathcal{R}} t$ (resp. $s \to^*_{\mathcal{R}} t$) iff there is a proof tree for $s \to t$ (resp. $s \to^* t$) using the inference system $\mathcal{I}(\mathcal{R})$, whose rules are *schematic* in the sense that each inference rule $\frac{B_1 \;\cdots\; B_n}{A}$ can be used under any *instance* $\frac{\sigma(B_1) \;\cdots\; \sigma(B_n)}{\sigma(A)}$ of the rule by a substitution σ.

$$\text{(R)}\quad \frac{}{x \to^* x} \qquad \text{(T)}\quad \frac{x \to y \qquad y \to^* z}{x \to^* z}$$

$$\text{(C)}_{f,i}\quad \frac{x_i \to y_i}{f(x_1,\ldots,x_i,\ldots,x_k) \to f(x_1,\ldots,y_i,\ldots,x_k)}\ \text{for all } f \in \mathcal{F}^{(k)} \text{ and } 1 \le i \le k \qquad \text{(Rl)}_\rho\quad \frac{s_1 \to^* t_1 \quad \cdots \quad s_n \to^* t_n}{\ell \to r}\ \text{for } \rho : \ell \to r \Leftarrow s_1 \approx t_1, \ldots, s_n \approx t_n \in \mathcal{R}$$

Figure 1: Inference rules for conditional rewriting with an oriented CTRS $\mathcal{R}$ with signature $\mathcal{F}$

*Partially supported by the EU (FEDER), and projects RTI2018-094403-B-C32, PROMETEO/2019/098, and SP20180225.

†Raúl Gutiérrez was also supported by INCIBE program "Ayudas para la excelencia de los equipos de investigación avanzada en ciberseguridad".

A first-order theory $\overline{\mathcal{R}}$ associated to $\mathcal{R}$, where $\to$ and $\to^*$ are seen as predicates, is obtained from $\mathcal{I}(\mathcal{R})$: the inference rules $\frac{B_1 \ \cdots \ B_n}{A}$ in $\mathcal{I}(\mathcal{R})$ are considered as *sentences* $(\forall x_1, \ldots, x_m)\, B_1 \wedge \cdots \wedge B_n \Rightarrow A$, where $\{x_1, \ldots, x_m\}$ is the (possibly empty) set of variables occurring in the atoms $B_1, \ldots, B_n$ and A. If such a set is empty, we write $B_1 \wedge \cdots \wedge B_n \Rightarrow A$.

Example 1. *Consider the following CTRS $\mathcal{R}$ (903.trs[1]):*

$$\begin{array}{rcl@{\qquad}rcl} le(0, s(y)) & \to & true & min(cons(x, nil)) & \to & x \\ le(s(x), s(y)) & \to & le(x, y) & min(cons(x, xs)) & \to & x \Leftarrow min(xs) \approx y, le(x, y) \approx true \\ le(x, 0) & \to & false & min(cons(x, xs)) & \to & y \Leftarrow min(xs) \approx y, le(x, y) \approx false \end{array}$$

The first-order theory $\overline{\mathcal{R}}$ for $\mathcal{R}$ is:

$$(\forall x)\ x \to^* x \quad (1) \qquad (\forall x, y, z)\ x \to y \Rightarrow min(x, z) \to min(y, z) \quad (8)$$
$$(\forall x, y, z)\ x \to y \wedge y \to z \Rightarrow x \to^* z \quad (2) \qquad (\forall x, y, z)\ x \to y \Rightarrow min(z, x) \to min(z, y) \quad (9)$$
$$(\forall x, y)\ x \to y \wedge x \to^* y \Rightarrow s(x) \to s(y) \quad (3) \qquad (\forall y)\ le(0, s(y)) \to true \quad (10)$$
$$(\forall x, y, z)\ x \to y \Rightarrow cons(x, z) \to cons(y, z) \quad (4) \qquad (\forall x, y)\ le(s(x), s(y)) \to le(x, y) \quad (11)$$
$$(\forall x, y, z)\ x \to y \Rightarrow cons(z, x) \to cons(z, y) \quad (5) \qquad (\forall x)\ le(x, 0) \to false \quad (12)$$
$$(\forall x, y, z)\ x \to y \Rightarrow le(x, z) \to le(y, z) \quad (6) \qquad (\forall x)\ min(cons(x, nil)) \to x \quad (13)$$
$$(\forall x, y, z)\ x \to y \Rightarrow le(z, x) \to le(z, y) \quad (7)$$
$$(\forall x, y, xs)\ min(xs) \to^* y \wedge le(x, y) \to^* true \Rightarrow min(cons(x, xs)) \to x \quad (14)$$
$$(\forall x, xs)\ min(xs) \to^* y \wedge le(x, y) \to^* false \Rightarrow min(cons(x, xs)) \to y \quad (15)$$

2 Feasibility Sequences

Given a CTRS $\mathcal{R}$ and terms s and t, we say that the atom $s \to^* t$ is *$\mathcal{R}$-feasible* (or just *feasible* if no confusion arises) if there is a substitution σ such that the *reachability test* $\sigma(s) \to^*_{\mathcal{R}} \sigma(t)$ succeeds. As in [5, Definition 2], sequences $\mathcal{G} = (s_i \to^* t_i)_{i=1}^n$, where $n > 0$, are called *feasibility sequences*.[2] We say that $\mathcal{G}$ is *$\mathcal{R}$-feasible* if there is a substitution such that $\sigma(s_i) \to^*_{\mathcal{R}} \sigma(t_i)$ holds for all $1 \le i \le n$; we call $\mathcal{G}$ *infeasible* otherwise. In [4, 5], we presented an approach to deal with infeasibility using a satisfiability criterion: a sequence $\mathcal{G}$ as above is *infeasible* if the first-order theory $\overline{\mathcal{R}}$ together with the *negation* of the sentence

$$(\exists \vec{x}) \bigwedge_{i=1}^{n} s_i \to^* t_i \qquad (16)$$

where $\vec{x}$ are the variables occurring in terms s_i and t_i for $1 \le i \le n$, is *satisfiable* by any interpretation $\mathcal{A}$ of the function and predicate symbols, i.e., $\mathcal{A} \models \overline{\mathcal{R}} \cup \{\neg(16)\}$ holds [5, Theorem 6]. Actually, as showed in [3], if (16) is a logical consequence of $\overline{\mathcal{R}}$ (i.e., $\overline{\mathcal{R}} \vdash (16)$ holds), then the feasibility of $\mathcal{G}$ is *proved*. Thus, this logical approach provides a sound and complete method to (dis)prove feasibility.

Example 2. *Continuing with Example 1, the sequence: $min(nil) \to^* x, le(y, x) \to^* true$ corresponds to the following first-order formula (16): $(\exists x, y)\, min(nil) \to^* x \wedge le(y, x) \to^* true$*

In this paper, we present infChecker, a tool for proving and disproving feasibility conditions taking advantage of this logical approach:

http://zenon.dsic.upv.es/infChecker/

[1] This problem belongs to the database COPS of confluence problems in http://cops.uibk.ac.at/

[2] In [5, Definition 2] atoms $s \to t$ are also considered in feasibility sequences.

3 Feasibility Framework

In order to automatically analyze whether a sequence $\mathcal{G}$ is *feasible* or *infeasible*, we describe a framework similar to the one presented in [1] for termination purposes. We define appropriate notions of (feasibility) *problem* and *processor* and show how to apply processors in order to prove or disprove feasibility.

Definition 3 (fProblem and fProcessor)**.** *An* fProblem τ *is a pair* $\tau = (\mathcal{R}, \mathcal{G})$*, where* $\mathcal{R}$ *is a CTRS and* $\mathcal{G}$ *is a sequence* $(s_i \to^* t_i)_{i=1}^n$*. The* fProblem τ *is* feasible *if* $\mathcal{G}$ *is* $\mathcal{R}$*-feasible; otherwise it is* infeasible*.*

An fProcessor P *is a partial function from fProblems into sets of fProblems. Alternatively, it can return* "no"*.* $\mathcal{D}om(\mathsf{P})$ *represents the domain of* P*, i.e., the set of fProblems* τ *that* P *is defined for.*

An fProcessor P *is* sound *if for all* $\tau \in \mathcal{D}om(\mathsf{P})$*,* τ *is feasible whenever* $\forall \tau' \in \mathsf{P}(\tau)$*,* τ' *is feasible.*

An fProcessor P *is* complete *if for all* $\tau \in \mathcal{D}om(\mathsf{P})$*,* τ *is infeasible whenever either* $\mathsf{P}(\tau) =$"no" *or* $\exists \tau' \in \mathsf{P}(\tau)$*, such that* τ' *is infeasible.*

Feasibility problems can be proved or disproved by using a proof tree as follows.

Theorem 4 (Feasibility Proof Tree)**.** *Let* τ *be an fProblem. A feasibility proof tree* $\mathcal{T}$ *for* τ *is a tree whose inner nodes are labeled with fProblems and the leaves may also be labeled with either* "yes" *or* "no"*. The root of* $\mathcal{T}$ *is labeled with* τ *and for every inner node* n *labeled with* τ'*, there is a processor* P *such that* $\mathsf{P} \in \mathcal{D}om(\mathsf{P})$ *and: (1) if* $\mathsf{P}(\tau') = \mathsf{no}$ *then* n *has just one child, labeled with*"no"*; (2) if* $\mathsf{P}(\tau') = \emptyset$ *then* n *has just one child, labeled with* "yes"*; and (3) if* $\mathsf{P}(\tau') = \{\tau_1, \ldots, \tau_k\}$ *with* $k > 0$*, then* n *has* k *children labeled with the fProblems* $\tau_1, \ldots, \tau_k$*.*

Theorem 5 (Feasibility Framework)**.** *Let* $\mathcal{R}$ *be an oriented CTRS,* $\mathcal{G}$ *be a feasibility sequence, and* $\mathcal{T}$ *be a feasibility proof tree for* $\tau_I = (\mathcal{R}, \mathcal{G})$*. Then: (1) if all leaves in* $\mathcal{T}$ *are labeled with* "yes" *and all involved processors are sound for the fProblems they are applied for, then* $\mathcal{G}$ *is* $\mathcal{R}$*-feasible; and (2) if* $\mathcal{T}$ *has a leaf labeled with* "no" *and all processors from* τ_I *to the leaf are complete for the fProblems they are applied for, then* $\mathcal{G}$ *is* $\mathcal{R}$*-infeasible.*

In the following subsections we describe a number of sound and complete fProcessors.

3.1 Satisfiability Processor

The following processor integrates the satisfiability approach described in [5] to prove infeasibility in our framework. When dealing with reachability, i.e., all the left-hand sides of the feasibility conditions in $\mathcal{G}$ are ground, we can restrict our theory to the set of usable rules. Given an fProblem $(\mathcal{R}, \mathcal{G})$, we let

$$\mathcal{U}(\mathcal{R}, \mathcal{G}) = \begin{cases} \bigcup_{s_i \to^* t_i \in \mathcal{G}} \mathcal{U}(\mathcal{R}, s_i) & \text{if all } s_i \text{ in } \mathcal{G} \text{ are ground} \\ R & \text{otherwise} \end{cases}$$

where, given a CTRS $\mathcal{R}$ and a term t, $\mathcal{U}(\mathcal{R}, t)$ are the usable rules $\mathcal{R}$ regarding t [5, Section 2].

Theorem 6 (Satisfiability Processor)**.** *Let* $\tau = (\mathcal{R}, \mathcal{G})$ *be an fProblem with* $\mathcal{G} = (s_i \to^* t_i)_{i=1}^n$*. Let* $\mathcal{A}$ *be a structure such that* $\mathcal{A} \neq \emptyset$ *and* $\mathcal{A} \models \overline{\mathcal{U}(\mathcal{R}, \mathcal{G})} \cup \{\neg(\exists \vec{x}) \bigwedge_{i=1}^n s_i \to^* t_i\}$*. The processor* $\mathsf{P}^{\mathsf{Sat}}$ *given by* $\mathsf{P}^{\mathsf{Sat}}(\tau) = \mathsf{no}$ *is sound and complete.*

In infChecker, we use the model generators AGES [2] and Mace4 [8] to find suitable structures $\mathcal{A}$ to be used in the implementation of $\mathsf{P}^{\mathsf{Sat}}$.

Example 7. *For $\mathcal{R}$ in Example 1 and $\mathcal{G}$ in Example 2, we obtain* $\mathsf{P}^{\mathsf{Sat}}(\tau_I) = \mathsf{no}$ *using* AGES[3].

3.2 Provability Processor

The following processor integrates the logic-based approach to program analysis described in [3] to prove feasibility by theorem proving.

Theorem 8 (Provability Processor)**.** *Let $\tau = (\mathcal{R}, \mathcal{G})$ be an fProblem with $\mathcal{G} = (s_i \to^* t_i)_{i=1}^n$ such that $\overline{\mathcal{R}} \vdash \ (\exists \vec{x})\ \bigwedge_{i=1}^n s_i \to^* t_i$ holds. The processor $\mathsf{P}^{\mathsf{Prov}}$ given by $\mathsf{P}^{\mathsf{Prov}}(\tau) = \emptyset$ is sound and complete.*

In infChecker, we use the theorem prover Prover9 [8] as a backend to implement $\mathsf{P}^{\mathsf{Prov}}$.

Example 9. *For $\mathcal{R}$ in Example 1 and $\mathcal{G} = le(x, min(y)) \to^* false, min(y) \to^* x$ (836.trs) with associated first-order formula (16) as follows:*

$$(\exists x, y)\ le(x, min(y)) \to^* false \wedge min(y) \to^* x \tag{17}$$

we have $\mathsf{P}^{\mathsf{Prov}}(\tau_I) = \emptyset$ *by using* Prover9[4].

3.3 Narrowing on Feasibility Conditions Processor

In the context of the 2D DP framework [7], there are powerful processors that can be applied to the conditions of the rules in order to simplify those conditions. We adapt the processor that narrow conditions to be used on fProblems.

Let $N_1(\mathcal{S}, s) = \{(t, \theta{\downarrow_{\mathcal{V}ar(s)}}) \mid s \rightsquigarrow_{\ell \to r \Leftarrow c, \theta} t, \ell \to r \Leftarrow c \in NRules(\mathcal{S}, s)\}$ represents the set of one-step $\mathcal{S}$-narrowings issued from s [7, Definition 79], where $NRules(\mathcal{S}, s)$ is the set of rules $\alpha : \ell \to r \Leftarrow c \in \mathcal{S}$ such that a nonvariable subterm t of s is a *narrex* of α, and $\theta{\downarrow_{\mathcal{V}ar(s)}}$ is a substitution defined by $\theta{\downarrow_{\mathcal{V}ar(s)}}\ (x) = \theta(x)$ if $x \in \mathcal{V}ar(s)$ and $\theta{\downarrow_{\mathcal{V}ar(s)}}\ (x) = x$ otherwise. As discussed in [7, Section 7.5], the set $N_1(\mathcal{S}, s)$ can be *infinite* if $NRules(\mathcal{S}, s)$ is *not* a TRS, i.e., it contains 'proper' conditional rules. In [7, Proposition 87] some sufficient conditions for finiteness of $N_1(\mathcal{S}, s)$ are given. Accordingly, we define a narrowing processor on fProblems. Given a feasibility sequence $\mathcal{G} = (s_i \to^* t_i)_{i=1}^n$ we let

$$\overline{\mathcal{N}}(\mathcal{S}, \mathcal{G}, i) = \{\mathcal{G}[\vec{\theta}, w \to^* t_i]_i \mid s_i \to^* t_i \in \mathcal{G}, (w, \theta) \in N_1(\mathcal{S}, s_i)\}$$

where $\vec{\theta}$ consists of new conditions $x_1 \to^* \theta(x_1), \ldots, x_m \to^* \theta(x_m)$ obtained from the bindings in θ for variables in $\mathcal{V}ar(s_i) = \{x_1, \ldots, x_m\}$.

Definition 10 (Narrowing on Feasibility Conditions Processor)**.** *Let $\tau = (\mathcal{R}, \mathcal{G})$ be an fProblem, $s_i \to^* t_i \in \mathcal{G}$, and $\mathcal{N} \subseteq \overline{\mathcal{N}}(\mathcal{R}, \mathcal{G}, i)$ finite.* P^{NC} *is given by* $\mathsf{P}^{\mathsf{NC}}(\tau) = \{(\mathcal{R}, \mathcal{N})\}$.

Theorem 11. P^{NC} *is sound. If $\mathcal{N} = \overline{\mathcal{N}}(\mathcal{R}, \mathcal{G}, i)$ and $s_i \to^* t_i \in \mathcal{G}$ is such that s_i and t_i do* not *unify and either s_i is* ground *or (1) $NRules(\mathcal{R}, s_i)$ is a TRS and (2) s_i is linear, then* P^{NC} *is complete.*

A complete example is given in the appendix.

[3] To ease readability, examples and details of the proofs (interpretations, etc.) are given in the appendix; due to space restrictions, they are not intended to be included in the final version.

[4] Proof is given in the appendix.

4 Experimental Evaluation

We participated in the *Infeasibility* (INF) category of the 2019 Confluence Competition (CoCo)[5]:

INF Tool	Yes	No	Total
infChecker	40	32	72
nonreach	30	0	30
Moca	26	0	26
maedmax	15	0	15
CO3	12	0	12

Note that answers *Yes/No* in the table refer to *infeasibility* problems (which is the focus of the competition). In our setting, given a CTRS $\mathcal{R}$ and an infeasibility problem given as a feasibility sequence $\mathcal{G}$, we just return *Yes* if τ_I is proved infeasible, and *No* if τ_I is proved feasible.

Apart from the 32 negative answers, there are 7 more examples that can be proved positively using infChecker only. Furthermore, there are 10 examples that can be proved by other tools and cannot be proved by infChecker.

5 Conclusions and Future Work

In this paper we present infChecker, a new tool for checking feasibility conditions of CTRSs that takes advantage of the logic-based approach presented in [3, 4, 5]. We succesfully participated in the 2019 Confluence Competition in the INF (infeasibility) category, being the most powerful tool for checking both infeasibility and feasibility.

Currently, the tool has only three processors. As a subject for future work, we would like to increase the power of the tool with new processors focused on feasibility conditions, analyze the examples that cannot be proved by infChecker but can be proved by other tools and extend the tool to deal with more involved rewrite systems (order-sorted, axioms...).

References

[1] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and Improving Dependency Pairs. *Journal of Automatic Reasoning*, 37(3):155–203, 2006.

[2] R. Gutiérrez and S. Lucas. Automatic Generation of Logical Models with AGES (System Description). In P. Fontaine, editor, *Proc. of the CADE 2019*, LNCS, to appear. Springer, 2019.

[3] S. Lucas. Proving Program Properties as First-Order Satisfiability. In F. Mesnard and P. J. Stuckey, editors, *LOPSTR 2018*, volume 11408 of *LNCS*, pages 3–21. Springer, 2019.

[4] S. Lucas and R. Gutiérrez. A Semantic Criterion for Proving Infeasibility in Conditional Rewriting. In B. Accattoli and B. Felgenhauer, editors, *Proc. of the IWC'17*, pages 15–19, 2017.

[5] S. Lucas and R. Gutiérrez. Use of Logical Models for Proving Infeasibility in Term Rewriting. *Information Processing Letters*, 136:90–95, 2018.

[6] S. Lucas, C. Marché, and J. Meseguer. Operational termination of conditional term rewriting systems. *Information Processing Letters*, 95(4):446–453, 2005.

[7] S. Lucas, J. Meseguer, and R. Gutiérrez. The 2D Dependency Pair Framework for conditional rewrite systems. Part I: Definition and basic processors. *Journal of Computer and System Sciences*, 96:74–106, 2018.

[8] W. McCune. Prover9 and Mace4. [online]. Available at https://www.cs.unm.edu/~mccune/mace4/.

[5] http://project-coco.uibk.ac.at/2019/

Residuals Revisited*

Christina Kohl and Aart Middeldorp

Department of Computer Science, University of Innsbruck, Austria
christina.kohl@uibk.ac.at,aart.middeldorp@uibk.ac.at

Abstract

Proof terms are a useful concept for comparing computations in term rewriting. The residual operation is an important operation on proof terms, used to define projection equivalence. We present a variant of the residual system (Definition 8.7.54 of TeReSe) that is (innermost) confluent and terminating, and thus can be used to decide projection equivalence.

1 Introduction

To reason about rewrite sequences in left-linear term rewrite systems, in [3, Chapter 8] and [4] de Vrijer and van Oostrom define and compare different notions of equivalence. In this paper we are concerned with one of these notions, projection equivalence, which is defined using residuals. We present a schematic rewrite system for computing residuals that operates on proof terms. The latter are used to represent rewrite sequences. Our rewrite system is a variant of the residual system defined in [3, Definition 8.7.54 and proof of Theorem 8.7.57] and [4, Definition 6.9 and proof of Theorem 6.12]. We identify several issues with the analysis in [3, 4] and propose a solution by imposing an evaluation strategy on the residual system. We establish (innermost) confluence and termination of the adapted system, and show how these properties are used to decide projection equivalence. The decision procedure is incorporated into ProTeM, a recent tool [1] for manipulating proof terms.

2 Proof Terms

Proof terms are built from function symbols, variables, and rule symbols as well as the binary *composition* operator ; which is used in infix notation. Rule symbols represent rewrite rules and have a fixed arity which is the number of different variables in the represented rule. We use Greek letters $(\alpha, \beta, \gamma, \dots)$ as rule symbols, and uppercase letters $(A, B, C, \dots)$ for proof terms.

If α is a rule symbol then lhs_α (rhs_α) denotes the left-hand (right-hand) side of the rewrite rule represented by α. Furthermore var_α denotes the list $(x_1, \dots, x_n)$ of variables appearing in α in some fixed order. The length of this list is the arity of α. Given a rule symbol α with $\mathsf{var}_\alpha = (x_1, \dots, x_n)$ and proof terms $A_1, \dots, A_n$, we write $\langle A_1, \dots, A_n \rangle_\alpha$ for the substitution $\{x_i \mapsto A_i \mid 1 \leqslant i \leqslant n\}$. A proof term A witnesses a rewrite sequence from its source $\mathsf{src}(A)$ to its target $\mathsf{tgt}(A)$, which are computed as follows:

$$\begin{aligned}
\mathsf{src}(x) &= \mathsf{tgt}(x) = x & \mathsf{src}(f(A_1, \dots, A_n)) &= f(\mathsf{src}(A_1), \dots, \mathsf{src}(A_n)) \\
\mathsf{src}(A\,;B) &= \mathsf{src}(A) & \mathsf{src}(\alpha(A_1, \dots, A_n)) &= \mathsf{lhs}_\alpha\langle \mathsf{src}(A_1), \dots, \mathsf{src}(A_n)\rangle_\alpha \\
\mathsf{tgt}(A\,;B) &= \mathsf{tgt}(B) & \mathsf{tgt}(f(A_1, \dots, A_n)) &= f(\mathsf{tgt}(A_1), \dots, \mathsf{tgt}(A_n)) \\
& & \mathsf{tgt}(\alpha(A_1, \dots, A_n)) &= \mathsf{rhs}_\alpha\langle \mathsf{tgt}(A_1), \dots, \mathsf{tgt}(A_n)\rangle_\alpha
\end{aligned}$$

*This research is supported by FWF (Austrian Science Fund) project P27528. An extended version of this paper will appear in the proceedings of CADE-27 [2].

Here f is an n-ary function symbol. We assume $\mathsf{tgt}(A) = \mathsf{src}(B)$ whenever the composition $A\,;B$ is used in a proof term. Proof terms A and B are *co-initial* if they have the same source.

Example 1. *Consider the TRS consisting of the rules* $\boldsymbol{\alpha}$, $\boldsymbol{\beta}$, $\boldsymbol{\gamma}$ *and the proof terms* A, B, C*:*

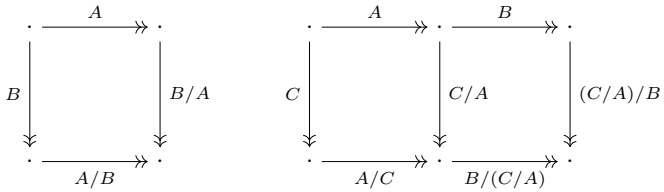
$$\boldsymbol{\alpha}\colon\ \mathsf{f}(\mathsf{a},x) \to \mathsf{g}(x,x) \qquad \boldsymbol{\beta}\colon\ \mathsf{a} \to \mathsf{b} \qquad \boldsymbol{\gamma}\colon\ \mathsf{b} \to \mathsf{c}$$
$$A = \boldsymbol{\alpha}(\boldsymbol{\beta}\,;\boldsymbol{\gamma}) \qquad B = (\boldsymbol{\alpha}(\mathsf{a})\,;\mathsf{g}(\boldsymbol{\beta},\boldsymbol{\beta}))\,;\mathsf{g}(\boldsymbol{\gamma},\boldsymbol{\gamma}) \qquad C = \mathsf{f}(\mathsf{a},\boldsymbol{\beta}\,;\boldsymbol{\gamma})\,;\boldsymbol{\alpha}(\mathsf{c})$$

We have $\mathsf{src}(A) = \mathsf{src}(B) = \mathsf{src}(C) = \mathsf{f}(\mathsf{a},\mathsf{a})$ *and* $\mathsf{tgt}(A) = \mathsf{tgt}(B) = \mathsf{tgt}(C) = \mathsf{g}(\mathsf{c},\mathsf{c})$*. The proof term* B *represents the sequence* $\mathsf{f}(\mathsf{a},\mathsf{a}) \to \mathsf{g}(\mathsf{a},\mathsf{a}) \twoheadrightarrow \mathsf{g}(\mathsf{b},\mathsf{b}) \twoheadrightarrow \mathsf{g}(\mathsf{c},\mathsf{c})$*.*

We can represent any rewrite sequence $\longrightarrow^*$ by a suitable proof term. A proof term without composition represents a multi-step, a proof term without composition and nested rule symbols represents a parallel step, and a proof term without composition and only one rule symbol represents a single step. If a proof term contains neither compositions nor rule symbols, it denotes an empty step.

3 Residuals

The residual operation computes, for co-initial proof terms A and B, which steps of A remain after performing B. The diagram on the left shows a desirable result of residuals and the diagram on the right provides the intuition behind equations (6) and (7) below:



In [3, Definition 8.7.54] and [4, Definition 6.9] the residual $A \,/\, B$ is defined by means of the following equations:

$$x \,/\, x = x \tag{1}$$
$$f(A_1,\dots,A_n) \,/\, f(B_1,\dots,B_n) = f(A_1 \,/\, B_1,\dots,A_n \,/\, B_n) \tag{2}$$
$$\alpha(A_1,\dots,A_n) \,/\, \alpha(B_1,\dots,B_n) = \mathsf{rhs}_\alpha\langle A_1 \,/\, B_1,\dots,A_n \,/\, B_n\rangle_\alpha \tag{3}$$
$$\alpha(A_1,\dots,A_n) \,/\, \mathsf{lhs}_\alpha\langle B_1,\dots,B_n\rangle_\alpha = \alpha(A_1 \,/\, B_1,\dots,A_n \,/\, B_n) \tag{4}$$
$$\mathsf{lhs}_\alpha\langle A_1,\dots,A_n\rangle_\alpha \,/\, \alpha(B_1,\dots,B_n) = \mathsf{rhs}_\alpha\langle A_1 \,/\, B_1,\dots,A_n \,/\, B_n\rangle_\alpha \tag{5}$$
$$C \,/\, (A\,;B) = (C \,/\, A) \,/\, B \tag{6}$$
$$(A\,;B) \,/\, C = (A \,/\, C)\,;(B \,/\, (C \,/\, A)) \tag{7}$$
$$A \,/\, B = \#(\mathsf{tgt}(B))^1 \qquad \text{(otherwise)}$$

Here A, B, C, $A_1,\dots,A_n$, $B_1,\dots,B_n$ are proof term *variables* that can be instantiated with arbitrary proof terms (so without /). The x in equation (1) denotes an *arbitrary* variable (in the underlying TRS), which cannot be instantiated.[2] For every rule α of the underlying TRS, the equation schemes (3)–(5) are suitably instantiated. For instance, for rule $\boldsymbol{\alpha}$ of Example 1

[1] In [3, 4] the wrong definition $A \,/\, B = \#(\mathsf{tgt}(A))$ is given.
[2] In [3, Remark 8.2.21] variables are treated as constants and (1) is absent.

we obtain the equations $\alpha(A) \,/\, \alpha(B) = \mathsf{g}(A \,/\, B, A \,/\, B)$, $\alpha(A) \,/\, \mathsf{f}(\mathsf{a}, B) = \alpha(A \,/\, B)$ and $\mathsf{f}(\mathsf{a}, A) \,/\, \alpha(B) = \mathsf{g}(A \,/\, B, A \,/\, B)$. In the final defining equation, $\#$ is the rule symbol of the special *error rule* $x \to \bot$. This rule is adopted to ensure that $A \,/\, B$ is defined for arbitrary left-linear TRSs. The defining equations are taken modulo

$$t \,;\, t \approx t \tag{8}$$

$$f(A_1, \ldots, A_n) \,;\, f(B_1, \ldots, B_n) \approx f(A_1 \,;\, B_1, \ldots, A_n \,;\, B_n) \tag{9}$$

The need for the so-called functorial identities (9) is explained in the following example (Vincent van Oostrom, personal communication).

Example 2. *Consider* $A = \mathsf{f}(\mathsf{g}(\beta) \,;\, \mathsf{g}(\gamma))$ *and* $B = \alpha(\mathsf{a})$ *in the TRS*

$$\alpha\colon\ \mathsf{f}(\mathsf{g}(x)) \to x \qquad \beta\colon\ \mathsf{a} \to \mathsf{b} \qquad \gamma\colon\ \mathsf{b} \to \mathsf{c}$$

When computing $A \,/\, B$ *without* (9)*, the* α*-instance* $\mathsf{f}(\mathsf{g}(A_1)) \,/\, \alpha(B_1) = A_1 \,/\, B_1$ *of schema* (4) *does not apply to* $A \,/\, B$ *since the* g *in* $\mathsf{f}(\mathsf{g}(A_1))$ *needs to be extracted from* $\mathsf{g}(\alpha) \,;\, \mathsf{g}(\gamma)$ *when computing* $A \,/\, B$*. As a consequence, the (otherwise) equation kicks in, producing the proof term* $\#(\mathsf{b})$ *that indicates an error. With* (9) *in place, the result of evaluating* $A \,/\, B$ *is the proof term* $\beta \,;\, \gamma$*, representing the desired sequence* $\mathsf{a} \to \mathsf{b} \to \mathsf{c}$*.*

It is not immediately clear that the defining equations on the preceding page constitute a well-defined definition of the residual operation. In [3, proof of Theorem 8.7.57] and [4, proof of Theorem 6.12] the defining equations together with (8) and (9) are oriented from left to right, resulting in a rewrite system $\mathcal{R}es$ that is claimed to be terminating and confluent. The residual of A over B is then defined as the unique normal form of $A \,/\, B$ in $\mathcal{R}es$.

There are two problems with this approach. First of all, when is the (otherwise) rule applied? In [3] this is not specified, resulting in an imprecise rewrite semantics of $\mathcal{R}es$. Keeping in mind that $A \,/\, B$ is supposed to be a total operation on *proof terms* (so no $/$ in A and B), a natural solution is to adopt an innermost evaluation strategy. This ensures that $C \,/\, A$ is evaluated before $(C \,/\, A) \,/\, B$ in the right-hand side of (6) and before $B \,/\, (C \,/\, A)$ in the right-hand side of (7). The (otherwise) condition is taken into account by imposing the additional restriction that the (otherwise) rule is applied to $A \,/\, B$ (with A and B in normal form) only if the other rules are not applicable. The second, and more serious, problem is that $\mathcal{R}es$ is *not* confluent.

Example 3. *Consider the TRS consisting of the rules*

$$\alpha\colon\ \mathsf{f}(x, y) \to \mathsf{f}(y, x) \qquad \beta\colon\ \mathsf{a} \to \mathsf{b} \qquad \gamma\colon\ \mathsf{f}(\mathsf{a}, x) \to x$$

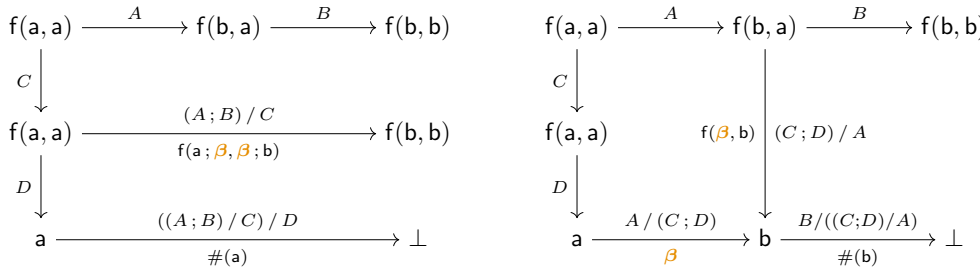
and the proof terms $A = f(\beta, \mathsf{a})$*,* $B = \alpha(\mathsf{b}, \beta)$*,* $C = \alpha(\mathsf{a}, \mathsf{a})$*, and* $D = \gamma(\mathsf{a})$*. There are two ways to compute* $(A \,;\, B) \,/\, (C \,;\, D)$*, starting with* (6) *or* (7)*:*

$$\begin{aligned}
((A \,;\, B) \,/\, C) \,/\, D &\to ((A \,/\, C) \,;\, (B \,/\, (C \,/\, A))) \,/\, D \\
&\to^* (\mathsf{f}(\mathsf{a} \,/\, \mathsf{a}, \beta \,/\, \mathsf{a}) \,;\, (B \,/\, \alpha(\mathsf{a} \,/\, \beta, \mathsf{a} \,/\, \mathsf{a}))) \,/\, D \\
&\to^* (\mathsf{f}(\mathsf{a}, \beta) \,;\, (B \,/\, \alpha(\mathsf{b}, \mathsf{a}))) \,/\, D \\
&\to (\mathsf{f}(\mathsf{a}, \beta) \,;\, \mathsf{f}(\beta \,/\, \mathsf{a}, \mathsf{b} \,/\, \mathsf{b})) \,/\, D \\
&\to^* (\mathsf{f}(\mathsf{a}, \beta) \,;\, \mathsf{f}(\beta, \mathsf{b})) \,/\, D \\
&\to \mathsf{f}(\mathsf{a} \,;\, \beta, \beta \,;\, \mathsf{b}) \,/\, D \to \#(\mathsf{a})
\end{aligned}$$

$$(A \,/\, (C \,;\, D)) \,;\, (B \,/\, ((C \,;\, D) \,/\, A))$$

$$
\begin{aligned}
&\to^* ((A \,/\, C) \,/\, D) \,;\, (B \,/\, ((C \,/\, A) \,;\, (D \,/\, (A \,/\, C))))\\
&\to^* (\mathsf{f}(\mathsf{a}, \beta) \,/\, D) \,;\, (B \,/\, (\alpha(\mathsf{b}, \mathsf{a}) \,;\, (D \,/\, \mathsf{f}(\mathsf{a}, \beta))))\\
&\to^* \beta \,;\, (B \,/\, (\alpha(\mathsf{b}, \mathsf{a}) \,;\, \gamma(\mathsf{b})))\\
&\to^* \beta \,;\, (\mathsf{f}(\beta, \mathsf{b}) \,/\, \gamma(\mathsf{b}))\\
&\to^* \beta \,;\, \#(\mathsf{b})
\end{aligned}
$$

The normal forms $\#(\mathsf{a})$ *and* $\beta \,;\, \#(\mathsf{b})$ *represent different failing computations:* $\mathsf{a} \to \bot$ *and* $\mathsf{a} \to \mathsf{b} \to \bot$*. The above computations are depicted in the diagrams below:*



To solve this problem we propose a drastic solution. When facing a term $A \,/\, B$ with A and B in normal form, the defining equations are evaluated from top to bottom and the first equation that matches is applied. This essentially means that the ambiguity between (6) and (7) is resolved by giving preference to the former. Due to innermost evaluation, no other critical situations arise. So we arrive at the following definition, where we turned equation (8) into rule (18), which is possible due to the presence of (19).

Definition 4. The *residual TRS* for proof terms consists of the following rules:

$$
\begin{aligned}
x \,/\, x &\to x && (10)\\
f(A_1, \ldots, A_n) \,/\, f(B_1, \ldots, B_n) &\to f(A_1 \,/\, B_1, \ldots, A_n \,/\, B_n) && (11)\\
\alpha(A_1, \ldots, A_n) \,/\, \alpha(B_1, \ldots, B_n) &\to \mathsf{rhs}_\alpha\langle A_1 \,/\, B_1, \ldots, A_n \,/\, B_n\rangle_\alpha && (12)\\
\alpha(A_1, \ldots, A_n) \,/\, \mathsf{lhs}_\alpha\langle B_1, \ldots, B_n\rangle_\alpha &\to \alpha(A_1 \,/\, B_1, \ldots, A_n \,/\, B_n) && (13)\\
\mathsf{lhs}_\alpha\langle A_1, \ldots, A_n\rangle_\alpha \,/\, \alpha(B_1, \ldots, B_n) &\to \mathsf{rhs}_\alpha\langle A_1 \,/\, B_1, \ldots, A_n \,/\, B_n\rangle_\alpha && (14)\\
C \,/\, (A \,;\, B) &\to (C \,/\, A) \,/\, B && (15)\\
(A \,;\, B) \,/\, C &\to (A \,/\, C) \,;\, (B \,/\, (C \,/\, A)) && (16)\\
A \,/\, B &\to \#(\mathsf{tgt}(B)) && (17)\\
x \,;\, x &\to x && (18)\\
f(A_1, \ldots, A_n) \,;\, f(B_1, \ldots, B_n) &\to f(A_1 \,;\, B_1, \ldots, A_n \,;\, B_n) && (19)
\end{aligned}
$$

We adopt innermost evaluation with the condition that the rules (10)–(17) are evaluated from top to bottom.

The residual TRS operates on *closed* proof terms, which are proof terms without proof term variables, to ensure that $\mathsf{tgt}(B)$ in the right-hand side of (17) can be evaluated. (Variables of the underlying TRS are allowed in proof terms.)

Example 5. *Consider the TRS of Example 1. For* $D = \alpha(\beta)$ *and* $E = \alpha(\mathsf{a}) \,;\, \mathsf{g}(\beta, \beta)$ *we have*

$$D \,/\, E = \alpha(\beta) \,/\, (\alpha(\mathsf{a}) \,;\, \mathsf{g}(\beta, \beta)) \to (\alpha(\beta) \,/\, \alpha(\mathsf{a})) \,/\, \mathsf{g}(\beta, \beta) \to \mathsf{g}(\beta \,/\, \mathsf{a}, \beta \,/\, \mathsf{a}) \,/\, \mathsf{g}(\beta, \beta)$$

$$\begin{aligned}
&\to^* \mathsf{g}(\beta,\beta) \,/\, \mathsf{g}(\beta,\beta) \to \mathsf{g}(\beta \,/\, \beta, \beta \,/\, \beta) \to^* \mathsf{g}(\mathsf{b},\mathsf{b})\\
E \,/\, D \;=\; &(\alpha(\mathsf{a}) \,;\, \mathsf{g}(\beta,\beta)) \,/\, \alpha(\beta) \to (\alpha(\mathsf{a}) \,/\, \alpha(\beta)) \,;\, (\mathsf{g}(\beta,\beta) \,/\, (\alpha(\beta) \,/\, \alpha(\mathsf{a})))\\
&\to^* \mathsf{g}(\mathsf{a} \,/\, \beta, \mathsf{a} \,/\, \beta) \,;\, (\mathsf{g}(\beta,\beta) \,/\, \mathsf{g}(\beta \,/\, \mathsf{a}, \beta \,/\, \mathsf{a})) \to^* \mathsf{g}(\mathsf{b},\mathsf{b}) \,;\, (\mathsf{g}(\beta,\beta) \,/\, \mathsf{g}(\beta,\beta))\\
&\to\; \mathsf{g}(\mathsf{b},\mathsf{b}) \,;\, \mathsf{g}(\beta \,/\, \beta, \beta \,/\, \beta) \to^* \mathsf{g}(\mathsf{b},\mathsf{b}) \,;\, \mathsf{g}(\mathsf{b},\mathsf{b}) \to \mathsf{g}(\mathsf{b} \,;\, \mathsf{b}, \mathsf{b} \,;\, \mathsf{b}) \to^* \mathsf{g}(\mathsf{b},\mathsf{b})
\end{aligned}$$

Lemma 6. *The residual TRS is terminating and confluent on closed proof terms.*

Proof. Confluence of the residual TRS is obvious because of the innermost evaluation strategy and the fact that there is no root overlap between its rules (due to the imposed evaluation order). Showing termination is non-trivial because of the nested occurrences of / in the right-hand sides of (15) and (16). As suggested in [3, Exercise 8.7.58] one can use semantic labeling [5]. We take the well-founded algebra $\mathcal{A}$ with carrier $\mathbb{N}$ equipped with the standard order $>$ and the following weakly monotone interpretation and labeling functions:

$$\begin{gathered}
\alpha_{\mathcal{A}}(x_1,\dots,x_n) = f_{\mathcal{A}}(x_1,\dots,x_n) = \max\{x_1,\dots,x_n\}\\
;_{\mathcal{A}}(x,y) = x + y + 1 \qquad /_{\mathcal{A}}(x,y) = x \qquad \#_{\mathcal{A}}(x) = \bot_{\mathcal{A}} = 0\\
L_{;} = L_f = L_\alpha = L_\# = L_\bot = \varnothing \qquad L_{/} = \mathbb{N} \qquad \mathsf{lab}_{/}(x,y) = x + y
\end{gathered}$$

The algebra $\mathcal{A}$ is a quasi-model of the residual TRS. Hence termination is a consequence of termination of its labeled version. The latter follows from LPO with well-founded precedence $/_i > /_j$ for all $i > j$ and $/_0 > \,; > f > \alpha > \# > \bot$ for all function symbols f and rule symbols α. □

The residual TRS is used to define projection equivalence.

Definition 7. The *projection order* $\lesssim$ and *projection equivalence* $\simeq$ are defined on co-initial proof terms as follows: $A \lesssim B$ if $A \,/\, B \to^* \mathsf{tgt}(B)$ and $A \simeq B$ if both $A \lesssim B$ and $B \lesssim A$.

Example 8. *The proof terms A, B, and C of Example 1 are projection equivalent since the residuals $A \,/\, B$, $B \,/\, A$, $A \,/\, C$, and $C \,/\, A$ all rewrite to the same normal form* $\mathsf{g}(\mathsf{c},\mathsf{c})$.

Lemma 6 provides us with an easy decision procedure for projection equivalence: $A \simeq B$ if and only the (unique) normal forms of $A \,/\, B$ and $B \,/\, A$ with respect to the residual TRS coincide and contain neither rule symbols nor compositions. This procedure is implemented in ProTeM[3] [1], a tool for manipulating proof terms. We refer to [2] for further details.

References

[1] Christina Kohl and Aart Middeldorp. ProTeM: A proof term manipulator (system description). In *Proc. 3rd FSCD*, volume 108 of *LIPIcs*, pages 31:1–31:8, 2018. doi: 10.4230/LIPIcs.FSCD.2018.31.

[2] Christina Kohl and Aart Middeldorp. Composing proof terms. In *Proc. 27th CADE*, LNAI, 2019. Accepted for publication.

[3] Terese, editor. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[4] Vincent van Oostrom and Roel de Vrijer. Four equivalent equivalences of reductions. In *Proc. 2nd WRS*, volume 70(6) of *ENTCS*, pages 21–61, 2002. doi: 10.1016/S1571-0661(04)80599-1.

[5] Hans Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24:89–105, 1995. doi: 10.3233/FI-1995-24124.

[3] http://informatik-protem.uibk.ac.at/

CoCo 2019

infChecker at the 2019 Confluence Competition*

Raúl Gutiérrez[1]† and Salvador Lucas[1]

ELP group, DSIC, Universitat Politècnica de València
Camino de Vera s/n, E-46022 Valencia, Spain
{rgutierrez,slucas}@dsic.upv.es

1 Overview

infChecker 1.0 is a tool for checking *(in)feasibility* of sequences $(s_i \to^* t_i)_{i=1}^n$ [3] over *Conditional Term Rewriting Systems* (CTRSs) based on a *Feasibility Framework* similar to the *Dependency Pair Framework* used in termination:

http://zenon.dsic.upv.es/infChecker/

infChecker is written in Haskell. Three *processors* have been implemented for the aforementioned feasibility framework:

- $\mathsf{P}^{\mathsf{Sat}}$ integrates the satisfiability approach described in [3] to prove infeasibility. In infChecker, we use the model generators AGES [1] and Mace4 [5] to find a proof.
- $\mathsf{P}^{\mathsf{Prov}}$ integrates the logic-based approach to program analysis described in [2] to prove feasibility by theorem proving. In infChecker, we use the theorem prover Prover9 [5].
- P^{NC} adapt the processor that narrow conditions in the 2D DP framework [4] to be used with feasibility sequences.

Since we have three processors, our proof strategy is very simple: (1) first, we try to prove feasibility using $\mathsf{P}^{\mathsf{Prov}}$; (2) if $\mathsf{P}^{\mathsf{Prov}}$ fails, we apply $\mathsf{P}^{\mathsf{Sat}}$; (3) if $\mathsf{P}^{\mathsf{Sat}}$ fails, we apply P^{NC}; (4) if P^{NC} succeeds and modifies the feasibility sequence, we go to (2), otherwise we return MAYBE.

We succesfully participated in the 2019 Confluence Competition in the INF (infeasibility) category, being the most powerful tool for proving and disproving infeasibility.

References

[1] R. Gutiérrez and S. Lucas. Automatic Generation of Logical Models with AGES (System Description). In *Proc. of CADE 2019*, LNCS to appear. Springer, 2019.

[2] S. Lucas. Proving Program Properties as First-Order Satisfiability. In Selected and Revised papers from *LOPSTR 2018* LNCS 11408:3–21. Springer, 2019.

[3] S. Lucas and R. Gutiérrez. Use of Logical Models for Proving Infeasibility in Term Rewriting. *Information Processing Letters*, 136:90–95, 2018.

[4] S. Lucas, J. Meseguer, and R. Gutiérrez. The 2D Dependency Pair Framework for conditional rewrite systems. Part I: Definition and basic processors. *Journal of Computer and System Sciences*, 96:74–106, 2018.

[5] W. McCune. Prover9 and Mace4. [online]. Available at https://www.cs.unm.edu/~mccune/mace4/.

*Partially supported by the EU (FEDER), and projects TIN2015-69175-C4-1-R, PROMETEO/2019/098, and SP20180225.

†Raúl Gutiérrez was also supported by INCIBE program "Ayudas para la excelencia de los equipos de investigación avanzada en ciberseguridad".

CO3 (Version 2.0)

Naoki Nishida and Yuya Maeda

Graduate School of Informatics, Nagoya University, Nagoya, Japan
nishida@i.nagoya-u.ac.jp yuya@trs.css.i.nagoya-u.ac.jp

CO3, a converter for proving confluence of conditional TRSs,[1] tries to prove confluence of conditional term rewriting systems (CTRSs, for short) by using a transformational approach (cf. [4]). The tool first transforms a given weakly-left-linear (WLL, for short) 3-DCTRS into an unconditional term rewriting system (TRS, for short) by using $\mathbb{U}_{conf}$ [2], a variant of the *unraveling* $\mathbb{U}$ [6], and then verifies confluence of the transformed TRS by using the following theorem: a 3-DCTRS $\mathcal{R}$ is confluent if $\mathcal{R}$ is WLL and $\mathbb{U}_{conf}(\mathcal{R})$ is confluent [1, 2]. The tool is very efficient because of very simple and lightweight functions to verify properties such as confluence and termination of TRSs. In the present version, a *narrowing-tree*-based approach [5, 3] to prove infeasiblity of a condition w.r.t. a specified CTRS has been implemented. The approach is applicable to *syntactically deterministic* CTRSs that are operationally terminating and *ultra-right-linear* w.r.t. the *optimized* unraveling.

To prove confluence by means of narrowing trees, the tool first computes the (conditional) critical pairs, and then proves their joinability as follows: a critical pair $\langle s, t\rangle \Leftarrow c$ is joinable if (1) c is the empty list and $s = t$, or (2) the narrowing tree for c can be simplified to a tree that defines the empty set of substitutions. For example, let us consider `489.trs` in Cops which is an operationally terminating normal 1-CTRS, and has a conditional critical pair $\langle \mathsf{true}, \mathsf{false}\rangle \Leftarrow \mathsf{o}(x) \twoheadrightarrow \mathsf{true}, \mathsf{e}(x) \twoheadrightarrow \mathsf{true}$. As a narrowing tree for condition $\mathsf{o}(x) \twoheadrightarrow \mathsf{true}, \mathsf{e}(x) \twoheadrightarrow \mathsf{true}$ w.r.t. `489.trs`, we construct the following production rules for a regular tree grammar [5]:

$$\begin{aligned}
\Gamma_{\mathsf{e}(x)\to\mathsf{true}\,\&\,\mathsf{o}(x)\to\mathsf{true}} &\to \textsc{Rec}(\Gamma_{\mathsf{e}(x')\to\mathsf{true}}, \{x \mapsto x'\}) \,\&\, \textsc{Rec}(\Gamma_{\mathsf{o}(x'')\to\mathsf{true}}, \{x \mapsto x''\})\\
\Gamma_{\mathsf{e}(x')\to\mathsf{true}} &\to id \,\&\{x' \mapsto \mathsf{0}\} \mid \big(\textsc{Rec}(\Gamma_{\mathsf{o}(x'')\to\mathsf{true}}, \{x_1 \mapsto x''\}) \,\&\, id\big) \,\&\{x' \mapsto \mathsf{s}(x_1)\}\\
&\quad \mid \big(\textsc{Rec}(\Gamma_{\mathsf{e}(x')\to\mathsf{true}}, \{x_2 \mapsto x'\}) \,\&\, \varnothing\big) \,\&\{x' \mapsto \mathsf{s}(x_2)\}\\
\Gamma_{\mathsf{o}(x'')\to\mathsf{true}} &\to \varnothing \,\&\{x'' \mapsto \mathsf{0}\} \mid \big(\textsc{Rec}(\Gamma_{\mathsf{e}(x')\to\mathsf{true}}, \{x_3 \mapsto x'\}) \,\&\, id\big) \,\&\{x'' \mapsto \mathsf{s}(x_3)\}\\
&\quad \mid \big(\textsc{Rec}(\Gamma_{\mathsf{o}(x'')\to\mathsf{true}}, \{x_4 \mapsto x''\}) \,\&\, \varnothing\big) \,\&\{x'' \mapsto \mathsf{s}(x_4)\}
\end{aligned}$$

These rules can be simplified to $\Gamma_{\mathsf{e}(x)\to\mathsf{true}\,\&\,\mathsf{o}(x)\to\mathsf{true}} \to \varnothing$, and the critical pair is infeasible.

To prove infeasibility of a condition c, the tool first proves confluence, and then linearizes c if failed to prove confluence. Then, the tool computes and simplifies a narrowing tree for c, and examines the emptiness of the narrowing tree.

References

[1] K. Gmeiner, B. Gramlich, and F. Schernhammer. On soundness conditions for unraveling deterministic conditional rewrite systems. In *Proc. RTA 2012*, vol. 15 of *LIPIcs*, pp. 193–208, 2012.

[2] K. Gmeiner, N. Nishida, and B. Gramlich. Proving confluence of conditional term rewriting systems via unravelings. In *Proc. IWC 2013*, pp. 35–39, 2013.

[3] Y. Maeda, N. Nishida, M. Sakai, and T. Kobayashi. Extending narrowing trees to basic narrowing in term rewriting. IEICE Tech. Rep. SS2018-39, Vol. 118, No. 385, pp. 73–78, 2019, in Japanese.

[4] N. Nishida, T. Kuroda, and K. Gmeiner. CO3 (Version 1.3). In *Proc. IWC 2016*, p. 74, 2016.

[5] N. Nishida and Y. Maeda. Narrowing trees for syntactically deterministic conditional term rewriting systems. In *Proc. FSCD 2018*, vol. 108 of *LIPIcs*, pp. 26:1–26:20, 2018.

[6] E. Ohlebusch. Termination of logic programs: Transformational methods revisited. *Appl. Algebra Eng. Commun. Comput.*, 12(1/2):73–116, 2001.

[1] http://www.trs.css.i.nagoya-u.ac.jp/co3/

ACP: System Description for CoCo 2019

Takahito Aoto[1] and Masaomi Yamaguchi[2]

[1] Institute of Science and Technology, Niigata University
aoto@ie.niigata-u.ac.jp
[2] Graduate School of Information Sciences, Tohoku University
masaomi.yamaguchi.t4@dc.tohoku.ac.jp

A primary functionality of ACP is proving confluence of term rewriting systems (TRSs). ACP integrates multiple direct criteria for guaranteeing confluence of TRSs. It also incorporates divide–and–conquer criteria by which confluence or non-confluence of TRSs can be inferred from those of their components. Several methods for disproving confluence are also employed. For some criteria, it supports generation of proofs in CPF format that can be certified by certifiers. The internal structure of the prover is kept simple and is mostly inherited from the version 0.11a, which has been described in [2].

This year we have added a decision procedure of UNC for shallow TRSs. The decidability of UNC for shallow TRSs has been shown in [4]; our new efficient procedure and a correctness proof are reported in [5]. We have also added a functionality to deal with commutation problems. Our (dis)proofs of commutation are based on a development closed criterion [6] and a simple search for counter examples. Lastly, we have also added a confluence checking using ordered rewriting [3].

ACP is written in Standard ML of New Jersey (SML/NJ) and the source code is also available from [1]. It uses a SAT prover such as MiniSAT and an SMT prover YICES as external provers. It internally contains an automated (relative) termination prover for TRSs but external (relative) termination provers can be substituted optionally. Users can specify criteria to be used so that each criterion or any combination of them can be tested. Several levels of verbosity are available for the output so that users can investigate details of the employed approximations for each criterion or can get only the final result of prover's attempt.

References

[1] ACP (Automated Confluence Prover). http://www.nue.ie.niigata-u.ac.jp/tools/acp/.

[2] T. Aoto, J. Yoshida, and Y. Toyama. Proving confluence of term rewriting system automatically. In *Proc. of 20th RTA*, volume 5595 of *LNCS*, pages 93–102. Springer-Verlag, 2009.

[3] U. Martin and T. Nipkow. Ordered rewriting and confluence. In *Proc. 10th CADE*, volume 449 of *LNAI*, pages 366–380. Springer-Verlag, 1990.

[4] N. R. Radcliffe, L. F. T. Moreas, and R. M. Verma. Uniqueness of normal forms for shallow term rewrite systems. *ACM Transactions on Computational Logic*, 18(2):17:1–17:20, 2017.

[5] M. Yamaguchi. An efficient decision procedure of the UN property for shallow term rewriting systems. Bachlor's thesis, Niigata University, 2019.

[6] J. Yoshida, T. Aoto, and Y. Toyama. Automating confluence check of term rewriting systems. *Computer Software*, 26(2):76–92, 2009.

AGCP: System Description for CoCo 2019

Takahito Aoto

Institute of Science and Technology, Niigata University
aoto@ie.niigata-u.ac.jp

AGCP (Automated Groud Confluence Prover) [1] is a tool for proving ground confluence of many-sorted term rewriting systems. AGCP is written in Standard ML of New Jersey (SML/NJ). AGCP proves ground confluence of many-sorted term rewriting systems based on two ingredients. One ingredient is to divide the ground confluence problem of a many-sorted term rewriting system $\mathcal{R}$ into that of $\mathcal{S} \subseteq \mathcal{R}$ and the inductive validity problem of equations $u \approx v$ w.r.t. $\mathcal{S}$ for each $u \to r \in \mathcal{R} \setminus \mathcal{S}$. Here, an equation $u \approx v$ is inductively valid w.r.t. $\mathcal{S}$ if all its ground instances $u\sigma \approx v\sigma$ is valid w.r.t. $\mathcal{S}$, i.e. $u\sigma \overset{*}{\leftrightarrow}_{\mathcal{S}} v\sigma$. Another ingredient is to prove ground confluence of a many-sorted term rewriting system via the *bounded ground convertibility* of the critical pairs. Here, an equation $u \approx v$ is said to be bounded ground convertibile w.r.t. a quasi-order $\succsim$ if $u\theta_g \overset{*}{\underset{\succsim}{\longleftrightarrow}}_{\mathcal{R}} v\theta_g$ for any its ground instance $u\sigma_g \approx v\sigma_g$, where $x \overset{*}{\underset{\succsim}{\longleftrightarrow}} y$ iff there exists $x = x_0 \leftrightarrow \cdots \leftrightarrow x_n = y$ such that $x \succsim x_i$ or $y \succsim x_i$ for every x_i.

Rewriting induction [3] is a well-known method for proving inductive validity of many-sorted term rewriting systems. In [1], an extension of rewriting induction to prove bounded ground convertibility of the equations has been reported. Namely, for a reduction quasi-order $\succsim$ and a quasi-reducible many-sorted term rewriting system $\mathcal{R}$ such that $\mathcal{R} \subseteq \succ$, the extension proves bounded ground convertibility of the input equations w.r.t. $\succsim$. The extension not only allows to deal with non-orientable equations but also with many-sorted TRSs having non-free constructors. Several methods that add wider flexibility to the this approach are given in [2]: when suitable rules are not presented in the input system, additional rewrite rules are constructed that supplement or replace existing rules in order to obtain a set of rules that is adequate for applying rewriting induction; and an extension of the system of [2] is used if if the input system contains non-orientable constructor rules. AGCP uses these extension of the rewriting induction to prove not only inductive validity of equations but also the bounded ground convertibility of the critical pairs. Finally, some methods to deal with disproving ground confluence are added as reported in [2].

No new ground (non-)confluence criterion has been incorporated from the one submitted for CoCo 2018.

References

[1] T. Aoto and Y. Toyama. Ground confluence prover based on rewriting induction. In *Proc. of 1st FSCD*, volume 52 of *LIPIcs*, pages 33:1–33:12. Schloss Dagstuhl, 2016.

[2] T. Aoto, Y. Toyama and Y. Kimura. Improving Rewriting Induction Approach for Proving Ground Confluence. In *Proc. of 2nd FSCD*, volume 84 of *LIPIcs*, pages 7:1–7:18. Schloss Dagstuhl, 2017.

[3] U.S. Reddy. Term rewriting induction. In *Proc. of CADE-10*, volume 449 of *LNAI*, pages 162–177. Springer-Verlag, 1990.

Noko-Leipzig at the 2019 Confluence Competition

Johannes Waldmann

Institut für Informatik, HTWK Leipzig, johannes.waldmann@htwk-leipzig.de

Noko-Leipzig is a confluence checker for string rewriting.

Noko-Leipzig implements a new method for proving non-joinability using arctically weighted automata [4], a generalisation of other methods using automata [1, 3]. In parallel, it checks local confluence and termination.

We found that even the basic method (no automata) is enough to answer 34 YES and 1401 NO for the 1541 string rewriting systems (SRS) from TPDB [2]. To get more interesting examples, we generated and filtered some random SRS, and submitted them for the Confluence Problems database. Among these are a few that can only be handled by the new method of weighted automata.

Noko-Leipzig uses the same code base as the Matchbox termination prover [5]. With competitor CSI [6], Noko-Leipzig shares the property that it rhymes with a TV series.

References

[1] T. Aoto. Disproving confluence of term rewriting systems by interpretation and ordering. In *Proc. 9th FroCoS*, volume 8152 of *LNCS (LNAI)*, pages 311–326, 2013.

[2] A. Yamada et al. The termination problems data base. http://www.termination-portal.org/wiki/TPDB, 2019.

[3] B. Felgenhauer and R. Thiemann. Reachability analysis with state-compatible automata. In *Proc. 8th LATA*, volume 8370 of *LNCS*, pages 347–359, 2013.

[4] B. Felgenhauer and J. Waldmann. Proving non-joinability using weakly monotone algebras. Submitted, 2019.

[5] J. Waldmann. The matchbox termination prover. https://gitlab.imn.htwk-leipzig.de/waldmann/pure-matchbox, 2019.

[6] H. Zankl, B. Felgenhauer, and A. Middeldorp. CSI – A confluence tool. In *Proc. 23rd CADE*, volume 6803 of *LNCS (LNAI)*, pages 499–505, 2011.

CoCo 2019 Participant: CSI 1.2.3*

Bertram Felgenhauer, Aart Middeldorp, and Fabian Mitterwallner

Department of Computer Science, University of Innsbruck, Austria

CSI is a strong automatic tool for (dis)proving confluence of first-order term rewrite systems (TRSs). It has been in development since 2010. Its name is derived from the Confluence of the rivers Sill and Inn in Innsbruck. The tool is available from

http://cl-informatik.uibk.ac.at/software/csi

under a LGPLv3 license. A detailed description of CSI can be found in [2]. Compared to last year's version, CSI 1.2.3 contains an implementation of the (inefficient) decision procedure for UNC of shallow rewrite systems by Radcliffe, Moraes and Verma [3]. In addition, CSI 1.2.3 contains an implementation of right-reducibility (no right-hand side of a rewrite rule is a normal form) [1] as a sufficient condition for NFP (and UNC and UNR by implication).

CSI participated in the categories CPF-TRS, NFP, SRS TRS, UNC, and UNR of CoCo 2019. It won the NFP and UNR categories, and in connection with CeTA, the CPF-TRS category. Somewhat surprisingly, CSI also won the new SRS category. CSI came in second behind ACP in the TRS and UNC categories.

References

[1] T. Aoto and Y. Toyama. Automated proofs of unique normal forms w.r.t. conversion for term rewriting systems. In *CoRR*, abs/1807.00940, 2018. http://arxiv.org/abs/1807.00940.

[2] J. Nagele, B. Felgenhauer, and A. Middeldorp. CSI: New evidence – A progress report. In *Proc. 26th International Conference on Automated Deduction*, volume 10395 of *Lecture Notes in Artificial Intelligence*, pages 385–397, 2017. doi: 10.1007/978-3-319-63046-5_24.

[3] N.R. Radcliffe, L.F.T. Moraes, and R.M. Verma. Uniqueness of normal forms for shallow term rewrite systems. *ACM Transactions on Computational Logic*, 18(2):17:1–17:20, 2017. doi: 10.1145/3060144.

*Supported by FWF (Austrian Science Fund) project P27528.

CoCo 2019 Participant: FORT 2.1*

Franziska Rapp[1] and Aart Middeldorp[2]

[1] Allgemeines Rechenzentrum Innsbruck, Austria
[2] Department of Computer Science, University of Innsbruck, Austria
aart.middeldorp@uibk.ac.at

FORT is a decision and synthesis tool for the first-order theory of rewriting for finite left-linear right-ground rewrite systems. It implements the decision procedure for this theory, which uses tree automata techniques and goes back to Dauchet and Tison [1]. In this theory confluence-related properties on ground terms are easily expressible. The basic functionality of FORT is described in [3] and in [4] we report on several extensions, including witness generation for existentially quantified variables in formulas and support for combinations of rewrite systems. The latter allows to express commutation, which is a new category (COM) in CoCo 2019 [2].

FORT 2.1 is implemented in Java. A command-line version of the tool can be downloaded from

http://cl-informatik.uibk.ac.at/software/FORT/

FORT participated in the following CoCo 2019 categories: COM, GCR, NFP, UNC, and UNR. Some of the YES/NO answers it produced in the NFP and UNR were out of reach for CSI. Surprisingly, FORT won the COM category because of incorrect answers by the other participating tools. We expect this does not happen again.

References

[1] M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. In *Proc. 5th IEEE Symposium on Logic in Computer Science*, pages 242–248, 1990. doi: 10.1109/LICS.1990.113750.

[2] A. Middeldorp, J. Nagele, and K. Shintani. Confluence Competition 2019. In *Proc. 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Part III)*, volume 11429 of *LNCS*, pages 25–40, 2019. doi: 10.1007/978-3-030-17502-3_2.

[3] F. Rapp and A. Middeldorp. Automating the first-order theory of left-linear right-ground term rewrite systems. In *Proc. 1st International Conference on Formal Structures for Computation and Deduction*, volume 52 of *Leibniz International Proceedings in Informatics*, pages 36:1–36:12, 2016. doi: 10.4230/LIPIcs.FSCD.2016.36.

[4] F. Rapp and A. Middeldorp. FORT 2.0. In *Proc. 9th International Joint Conference on Automated Reasoning*, volume 10900 of *LNCS (LNAI)*, pages 81–88, 2018. doi: 10.1007/978-3-319-94205-6_6.

*Supported by FWF (Austrian Science Fund) project P30301.

Moca 0.1: A First-Order Theorem Prover for Horn Clauses

Yusuke Oi and Nao Hirokawa

JAIST, Japan

Moca is a fully automatic first-order theorem prover for Horn clauses. The tool, written in Haskell, is freely available from:

http://www.jaist.ac.jp/project/maxcomp/

The usage is: `moca.sh <file>`. Given a satisfiability problem in the TPTP CNF format [3], the tool outputs `Satisfiable` or `Unsatisfiable` if its satisfiability or unsatisfiability is proved, respectively, and `Maybe` otherwise. Given an infeasibility problem in the CoCo format [2], the tool outputs `YES` if its infeasibility is proved, and `MAYBE` otherwise.

Moca implements *maximal ordered completion* [4] and new *approximation* techniques. With a small example we illustrate how Moca uses them to solve problems. Consider the infeasibility problem of the conversion $x - x \leftrightarrow^* \mathsf{s}(x)$ for the TRS:

$$x - \mathsf{0} \to x \qquad \mathsf{0} - x \to \mathsf{0} \qquad \mathsf{s}(x) - \mathsf{s}(y) \to x - y$$

The problem can be regarded as the satisfiability problem of the Horn clauses:

$$x - \mathsf{0} \approx x \qquad \mathsf{0} - x \approx \mathsf{0} \qquad \mathsf{s}(x) - \mathsf{s}(y) \approx x - y \qquad x - x \not\approx \mathsf{s}(x)$$

By applying the *split-if* encoding [1] the problem reduces to the word problem of deciding $\mathsf{T} \not\approx_{\mathcal{E}} \mathsf{F}$ for the equational system $\mathcal{E}$:

$$x - \mathsf{0} \approx x \quad \mathsf{0} - x \approx \mathsf{0} \quad \mathsf{s}(x) - \mathsf{s}(y) \approx x - y \quad \mathsf{f}(\mathsf{s}(x), x) \approx \mathsf{F} \quad \mathsf{f}(x - x, x) \approx \mathsf{T}$$

In order to solve it our tool attempts to construct a ground-complete presentation of $\mathcal{E}$ by using maximal ordered completion. However, the attempt is doomed to fail as the completion diverges. Moca overcomes the divergence by approximating the last equation to the more general equation $\mathsf{f}(x - x, y) \approx \mathsf{T}$. This results in the following equational system:

$$x - \mathsf{0} \approx x \quad \mathsf{0} - x \approx \mathsf{0} \quad \mathsf{s}(x) - \mathsf{s}(y) \approx x - y \quad \mathsf{f}(\mathsf{s}(x), x) \approx \mathsf{F} \quad \mathsf{f}(x - x, y) \approx \mathsf{T}$$

Now maximal ordered completion builds up the finite ground-complete presentation $\mathcal{R}$ of the approximated equational system:

$$\begin{array}{lll} x - \mathsf{0} \to x & \mathsf{0} - x \to \mathsf{0} & \mathsf{s}(x) - \mathsf{s}(y) \to x - y \\ \mathsf{f}(\mathsf{0}, y) \to \mathsf{T} & \mathsf{f}(\mathsf{s}(x), x) \to \mathsf{F} & \mathsf{f}(x - x, y) \to \mathsf{T} \end{array}$$

Since $\mathsf{T}{\downarrow_{\mathcal{R}}} \neq \mathsf{F}{\downarrow_{\mathcal{R}}}$ holds, infeasibility of the conversion $x - x \leftrightarrow^* \mathsf{s}(x)$ is concluded. Details of Moca and its underlying methods will be presented in our forthcoming paper.

References

[1] K. Claessen and N. Smallbone. Efficient Encodings of First-Order Horn Formulas in Equational Logic. In *Proc. 9th IJCAR*, volume 10900 of *LNCS*, pages 388–404, 2018.

[2] A. Middeldorp, J. Nagele, and K. Shintani. Confluence Competition 2019. In *Proc. 25th TACAS (Part III)*, volume 11429 of *LNCS*, pages 25–40, 2019.

[3] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: From CNF to TH0, TPTP v6.4.0. *Journal of Automated Reasoning*, 59(4):483–502, 2017.

[4] S. Winkler and G. Moser. MædMax: A Maximal Ordered Completion Tool. In *Proc. 9th IJCAR*, volume 10900 of *LNCS*, pages 472–480, 2018.

CoLL-Saigawa 1.3: A Joint Confluence Tool*

Kiraku Shintani and Nao Hirokawa

JAIST, Japan

CoLL-Saigawa is a tool for automatically proving or disproving confluence of (ordinary) term rewrite systems (TRSs). The tool, written in OCaml, is freely available at:

http://www.jaist.ac.jp/project/saigawa/

The typical usage is: `collsaigawa <file>`. Here the input file is written in the TRS format [11]. The tool outputs `YES` if confluence of the input TRS is proved, `NO` if non-confluence is shown, and `MAYBE` if the tool does not reach any conclusion.

CoLL-Saigawa is a joint confluence tool of CoLL v1.3 [7] and Saigawa v1.9 [3]. If an input TRS is left-linear, CoLL proves confluence. Otherwise, Saigawa analyzes confluence. CoLL is a commutation tool specialized for left-linear TRSs. It proves confluence as self-commutation by using Hindley's commutation theorem [2] together with the three commutation criteria: Development closeness [1, 8], rule labeling with weight function [9], and Church-Rosser modulo A/C [5]. Saigawa can deal with non-left-linear TRSs. The tool employs the four confluence criteria: The criteria based on critical pair systems [4, Theorem 3] and on extended critical pairs [6, Theorem 2], rule labeling [9], and Church-Rosser modulo AC [5].

In this version (version 1.3) we attempted to rectify a bug in AC unification (see [10]), but it turned out that the bug still remains in the current implementation. We will correct it soon.

References

[1] T. Aoto, J. Yoshida, and Y. Toyama. Proving confluence of term rewriting systems automatically. In *Proc. 20th RTA*, volume 5595 of *LNCS*, pages 93–102, 2009.

[2] J. R. Hindley. *The Church-Rosser Property and a Result in Combinatory Logic*. PhD thesis, University of Newcastle-upon-Tyne, 1964.

[3] N. Hirokawa. Saigawa: A confluence tool. In *3rd Confluence Competition*, pages 1–1, 2014.

[4] N. Hirokawa and A. Middeldorp. Commutation via relative termination. In *Proc. 2nd IWC*, pages 29–33, 2013.

[5] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal on Computing*, 15(4):1155–1194, 1986.

[6] D. Klein and N. Hirokawa. Confluence of non-left-linear TRSs via relative termination. In *Proc. 18th LPAR*, volume 7180 of *LNCS*, pages 258–273, 2012.

[7] K. Shintani and N. Hirokawa. CoLL: A confluence tool for left-linear term rewrite systems. In *Proc. 25th CADE*, volume 9195 of *LNAI*, pages 127–136, 2015.

[8] V. van Oostrom. Developing developments. *Theoretical Computer Science*, 175(1):159–181, 1997.

[9] V. van Oostrom. Confluence by decreasing diagrams converted. In A. Voronkov, editor, *Proc. 19th RTA*, volume 5117 of *LNCS*, pages 306–320, 2008.

[10] J. Nagele, B. Felgenhauer, and A. Middeldorp. CSI: New evidence – a progress report. *Proc. 26th CADE*, volume 10395 of *LNAI*, pages 385–397, 2017.

[11] A. Middeldorp, J. Nagele, and K. Shintani. Confluence Competition 2019. *Proc. 25th TACAS*, volume 11429 of *LNCS*, pages 25–40, 2019.

*Supported by JSPS KAKENHI Grant Number 17K00011.

CoLL 1.3: A Commutation Tool

Kiraku Shintani

JAIST, Japan
s1820017@jaist.ac.jp

CoLL (version 1.3) is a tool for automatically proving commutation of left-linear term rewrite systems (TRSs). The tool, written in OCaml, is freely available at:

http://www.jaist.ac.jp/project/saigawa/coll/

The typical usage is: `coll <file>`. Here the input file is written in the commutation problem format [10]. The tool outputs `YES` if commutation of the input TRSs is proved, `NO` if non-commutation is shown, and `MAYBE` if the tool does not reach any conclusion.

In this tool commutation of left-linear TRSs is shown by *Hindley's Commutation Theorem*:

Theorem 1 ([3])**.** *ARSs* $\mathcal{A} = \langle A, \{\to_\alpha\}_{\alpha\in I}\rangle$ *and* $\mathcal{B} = \langle A, \{\to_\beta\}_{\beta\in J}\rangle$ *commute if* $\to_\alpha$ *and* $\to_\beta$ *commute for all* $\alpha \in I$ *and* $\beta \in J$*.*

Here indexes are interpreted as subsystems of the input TRSs. For every pair of subsystems the tool proves the commutation property, employing the three criteria: Development closeness [2, 7], rule labeling with weight function [8, 1], and Church-Rosser modulo A/C [4]. A detailed description of CoLL can be found in [6].

In this version (version 1.3) we attempted to rectify a bug in AC unification (see [9]), but it turned out that the bug still remains in the current implementation. We will correct it soon.

References

[1] T. Aoto. Automated confluence proof by decreasing diagrams based on rule-labelling. In *Proc. 21st RTA*, volume 6 of *LIPIcs*, pages 7–16, 2010.

[2] T. Aoto, J. Yoshida, and Y. Toyama. Proving confluence of term rewriting systems automatically. In *Proc. 20th RTA*, volume 5595 of *LNCS*, pages 93–102, 2009.

[3] J. R. Hindley. *The Church-Rosser Property and a Result in Combinatory Logic*. PhD thesis, University of Newcastle-upon-Tyne, 1964.

[4] J.-P. Jouannaud and H. Kirchner. Completion of a set of rules modulo a set of equations. *SIAM Journal on Computing*, 15(4):1155–1194, 1986.

[5] B. K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM*, 20:160–187, 1973.

[6] K. Shintani and N. Hirokawa. CoLL: A confluence tool for left-linear term rewrite systems. In *Proc. 25th CADE*, volume 9195 of *LNAI*, pages 127–136, 2015.

[7] V. van Oostrom. Developing developments. *Theoretical Computer Science*, 175(1):159–181, 1997.

[8] V. van Oostrom. Confluence by decreasing diagrams converted. In A. Voronkov, editor, *Proc. 19th RTA*, volume 5117 of *LNCS*, pages 306–320, 2008.

[9] J. Nagele, B. Felgenhauer, and A. Middeldorp. CSI: New evidence – a progress report. *Proc. 26th CADE*, volume 10395 of *LNAI*, pages 385–397, 2017.

[10] A. Middeldorp, J. Nagele, and K. Shintani. Confluence Competition 2019. *Proc. 25th TACAS*, volume 11429 of *LNCS*, pages 25–40, 2019.

CoCo 2019 Participant: CSIˆho 0.3.2

Julian Nagele

School of Electronic Engineering and Computer Science, Queen Mary University of London, UK
j.nagele@qmul.ac.uk

CSIˆho is a tool for automatically (dis)proving confluence of higher-order rewrite systems, specifically pattern rewrite systems (PRSs) as introduced by Nipkow [3, 7]. CSIˆho focuses on patterns in order to ensure decidability of unification for computing critical pairs. To this end CSIˆho implements a version of Nipkow's algorithm for higher-order pattern unification [8]. CSIˆho is an extension of CSI, a confluence prover for first-order rewrite systems. The tool is available at

http://cl-informatik.uibk.ac.at/software/csi/ho

Below we briefly list the criteria implemented by CSIˆho—a more detailed description of both CSIˆho and CSI can be found in [5, 6].

For terminating PRSs CSIˆho decides confluence by checking joinability of critical pairs [7]. As termination criteria CSIˆho implements a basic higher-order recursive path ordering and static dependency pairs with dependency graph decomposition and the subterm criterion. Alternatively, one can also use an external termination tool like WANDA [2] as an oracle. For potentially non-terminating systems CSIˆho supports weak orthogonality [10] and van Oostrom's result on development closed critical pairs [9]. As a divide-and-conquer technique CSIˆho implements modularity for left-linear PRSs—note that confluence of PRSs is not modular in general [1]. Moreover CSIˆho uses the simple technique of adding and removing redundant rules [4], adapted for PRSs.

No new features were added to CSIˆho since CoCo 2018. It ran unopposed in the HRS category of CoCo 2019.

References

[1] C. Appel, V. van Oostrom, and J. G. Simonsen. Higher-order (non-)modularity. In *Proc. 21st RTA*, volume 6 of *LIPIcs*, pages 17–32, 2010.

[2] Cynthia Kop. *Higher Order Termination*. PhD thesis, Vrije Universiteit, Amsterdam, 2012.

[3] R. Mayr and T. Nipkow. Higher-order rewrite systems and their confluence. *TCS*, 192(1):3–29, 1998.

[4] J. Nagele, B. Felgenhauer, and A. Middeldorp. Improving automatic confluence analysis of rewrite systems by redundant rules. In *Proc. 26th RTA*, volume 36 of *LIPIcs*, pages 257–268, 2015.

[5] Julian Nagele. *Mechanizing Confluence: Automated and Certified Analysis of First- and Higher-Order Rewrite Systems*. PhD thesis, University of Innsbruck, 2017.

[6] Julian Nagele, Bertram Felgenhauer, and Aart Middeldorp. CSI: New evidence — A progress report. In *Proc. 26th CADE*, volume 10395 of *LNCS (LNAI)*, pages 385–397, 2017.

[7] T. Nipkow. Higher-order critical pairs. In *Proc. 6th LICS*, pages 342–349, 1991.

[8] Tobias Nipkow. Functional unification of higher-order patterns. In *Proc. 8th LICS*, pages 64–74, 1993.

[9] V. van Oostrom. Developing developments. *TCS*, 175(1):159–181, 1997.

[10] V. van Oostrom and F. van Raamsdonk. Weak orthogonality implies confluence: The higher order case. In *Proc. 3rd LFCS*, volume 813 of *LNCS*, pages 379–392, 1994.

CoCo 2019 Participant: ConCon 1.9*

Christian Sternagel and Sarah Winkler

Department of Computer Science, University of Innsbruck, Austria

ConCon is a fully automatic confluence checker for *oriented* first-order conditional term rewrite systems (CTRSs). It is written in Scala and available under the LGPL license at

http://cl-informatik.uibk.ac.at/software/concon

For more details on its implementation and employed methods we refer to an earlier system description [2].

Apart from some refactoring to cater for the new INF category (for infeasibility) of CoCo the most significant new feature in ConCon 1.9 is its use of the external ordered completion tool MædMax [4] for proving infeasibility. This new technique comes with certificate generation and can be certified [1] by CeTA [3] since version 2.36.

CoCo 2019. Unfortunately, the above mentioned refactoring did have its price: In the *Confluence Competition* 2019 ConCon 1.9 had YES/NO conflicts (on Cops #869, #870, #854, #874, #858, #875, and #909) with the tool infChecker in the new INF category. Moreover, we noticed that despite there being no conflicts, there were answers in the CTRS category that we could not reproduce with the bugfix version 1.9.1 of ConCon. Therefore, ConCon dropped out of both of the above categories. (The problem was a flipped Boolean flag in the *exact tree automata completion* method that was inadvertently introduced during refactoring.)

On the one hand, this clearly shows the need for certification. On the other hand, it may be interesting to note, that in the *certified* CPF-CTRS category (were ConCon+CeTA was the only participant this year) ConCon could prove (non)confluence of 1.3 times as many CTRSs than the winner of the *non certified* CTRS category.

References

[1] C. Sternagel and S. Winkler Certified Equational Reasoning via Ordered Completion In *Proc. 27th CADE*, 2019, to appear.

[2] T. Sternagel, C. Sternagel, and A. Middeldorp CoCo Participant 2018: ConCon 1.5. In *Proc. 7th IWC*, 2018. http://cl-informatik.uibk.ac.at/events/iwc-2018/iwc2018.pdf

[3] R. Thiemann and C. Sternagel Certification of Termination Proofs using CeTA. In *Proc. 22nd TPHOLs*, volume 5674 of *LNCS*, pages 452–468, 2009. doi: doi:10.1007/978-3-642-03359-9_31.

[4] S. Winkler and G. Moser MædMax: A Maximal Ordered Completion Tool In *Proc. 9th IJCAR*, volume 10900 of *LNCS*, pages 472–480, 2018. doi: 10.1007/978-3-319-94205-6_31.

*Supported by FWF (Austrian Science Fund) project P27502.

CoCo 2019 Participant: CeTA 2.36*

Christian Sternagel and Sarah Winkler

Department of Computer Science, University of Innsbruck, Austria

The tool CeTA [4] is a certifier for, among other properties, (non-)confluence of term rewrite systems with and without conditions. Its soundness is proven as part of the formal proof library IsaFoR, the *Isabelle Formalization of Rewriting*.

In the following, we describe what is new in version 2.36 of CeTA. For further details we refer to the certification problem format (CPF) and the IsaFoR/CeTA website:

http://cl-informatik.uibk.ac.at/ceta/

Conditional term rewriting. Since version 2.36, CeTA supports the certification of ordered completion proofs for infeasibility of conditional rules and critical pairs [2].

CoCo 2019. In the 2019 edition of the Confluence Competition, CeTA was used by three tools in two categories to certify their generated proofs: by ACP [1] and CSI [5] in the CPF-TRS category, and by ConCon [3] in the CPF-CTRS category.

References

[1] ACP (Automated Confluence Prover). http://www.ie.riec.tohoku.ac.jp/tools/acp/.

[2] C. Sternagel and S. Winkler Certified Equational Reasoning via Ordered Completion In *Proc. 27th CADE*, 2019, to appear.

[3] T. Sternagel and A. Middeldorp. Conditional Confluence (System Description). In *Proc. 1st RTA-TLCA*, volume 8560 of *LNCS*, pages 456–465, 2014. doi: 10.1007/978-3-319-08918-8_31.

[4] R. Thiemann and C. Sternagel Certification of Termination Proofs using CeTA. In *Proc. 22nd TPHOLs*, volume 5674 of *LNCS*, pages 452–468, 2009. doi: doi:10.1007/978-3-642-03359-9_31.

[5] H. Zankl, B. Felgenhauer, and A. Middeldorp CSI – A Confluence Tool In *Proc. 23rd CADE*, volume 6803 of *LNCS*, pages 499–505, 2011. doi: 10.1007/978-3-642-22438-6_38.

*Supported by Austrian Science Fund (FWF), projects P27502, P27528, and T789.

CoCo 2019 Participant: nonreach*

Florian Meßner and Christian Sternagel

University of Innsbruck, Innsbruck, Austria
{florian.g.messner,christian.sternagel}@uibk.ac.at

The tool nonreach is an automated, efficient tool to check infeasibility with respect to oriented conditional term rewrite systems (CTRSs). The Haskell source code can be obtained from a public *git* repository hosted on *bitbucket*:

https://bitbucket.org/fmessner/nonreach

Given a CTRS (or a TRS) and one or more infeasibility problems, nonreach uses a combination of the following two approaches:

- *Decomposition* is used to split a problem into conjunctions of easier and disjunctions of more specific subproblems. This creates a tree structure.
- *Fast checks* are then used to prove leaves of the tree infeasible and simplify the structure.

These methods are applied alternately until either infeasibility was proven (by simplifying the tree to False) or a user-defined threshold of iterations has been reached (and nonreach concludes MAYBE).

Our decomposition methods are based on narrowing (with some heuristics) and proving root-nonreachability [2]. The fast checks are based on etcap [3] and the *inductive symbol transition graph* [2].

In the 2019 edition of the Confluence Competition nonreach took part in the *infeasibility* (INF) category and earned the second place. Additionally, nonreach was the second fastest tool. The fastest tool CO3,[1] however, only solved 12 problems, where nonreach solved 30. Furthermore, nonreach only required 0.35% of the time taken by the winner of the competition, infChecker.[2]

For more details concerning the implementation and usage of nonreach, we refer to the tool demonstration paper published in TACAS 2019 [1].

References

[1] Florian Meßner and Christian Sternagel. nonreach - A tool for nonreachability analysis. In *Proc. 25th TACAS*, pages 337–343, 2019. doi:10.1007/978-3-030-17462-0_19.

[2] Christian Sternagel and Akihisa Yamada. Reachability analysis for termination and confluence of rewriting. In *Proc. 25th TACAS*, pages 262–278, 2019. doi:10.1007/978-3-030-17462-0_15.

[3] René Thiemann and Christian Sternagel. Certification of Termination Proofs using CeTA. In *Proc. 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *LNCS*, pages 452–468. Springer, 2009. doi:10.1007/978-3-642-03359-9_31.

*This work is supported by the Austrian Science Fund (FWF): project P27502.

[1] https://www.trs.css.i.nagoya-u.ac.jp/co3/

[2] http://zenon.dsic.upv.es/infChecker/

CoCo 2019 Participant: MædMax 1.7*

Sarah Winkler

Department of Computer Science, University of Innsbruck, Austria

MædMax is a completion tool: given a set of first-order equations $\mathcal{E}$ as input, it performs standard, ordered, or normalized completion in the attempt to derive a (ground) complete presentation of $\mathcal{E}$. It can also act as an equational theorem prover: if in addition to $\mathcal{E}$ a goal equation $s \approx t$ is provided as input, it will check whether there is a substitution σ such that $s\sigma \leftrightarrow^*_{\mathcal{E}} t\sigma$ holds.

In contrast to traditional completion tools, MædMax implements *maximal completion*. It is thus fully automatic in that no reduction order is required as input; instead, a suitable orientation of equations is detected by solving a maxSMT optimization problem. Details on this approach can be found in the initial proposal of maximal completion by Klein and Hirokawa [1] and a recent system description of MædMax [3].

MædMax is written in OCaml and available under the BSD license at

http://cl-informatik.uibk.ac.at/software/maedmax

CoCo 2019. In version 1.7 released for CoCo 2019 an infeasibility mode was added to participate in the new INF category. In this mode MædMax uses only one single technique to establish infeasibility, employing the above mentioned theorem proving capabilities. We briefly outline the idea.

Suppose $\mathcal{R}$ is the unconditional TRS of a given CTRS $\mathcal{C}$, and the condition c is given as a sequence of pairs of terms $s_1 \approx t_1, \ldots, s_k \approx t_k$. Then c is infeasible whenever there is no substitution σ such that $s_i\sigma \to^*_{\mathcal{C}} t_i\sigma$ holds for all $1 \leqslant i \leqslant k$ (in case of an *oriented* CTRS). Now, it is obviously a sound overapproximation to ensure that there is no σ such that $s_i\sigma \leftrightarrow^*_{\mathcal{R}} t_i\sigma$ for all $1 \leq i \leq k$. Thus MædMax uses a fresh function symbol c and attempts to decide the goal $\mathsf{c}(s_1, \ldots, s_k) \approx \mathsf{c}(t_1, \ldots, t_k)$ with respect to $\mathcal{R}$ considered as a set of input equalities. This task can be achieved using the aforementioned equational theorem proving techniques. Moreover, MædMax can output an XML certificate for such an infeasibility proof which is checkable by the proof checker CeTA. Further details can be found in [2].

Since this criterion for infeasibility constitutes an overapproximation, MædMax can only return YES answers in its infeasibility mode. The results of CoCo 2019 show that the supported technique is not very powerful compared to other tools. However, an upside of the approach is that it can also be used for CTRSs with other condition types than *oriented*.

References

[1] D. Klein and N. Hirokawa. Maximal completion. In *Proc. 22nd RTA*, volume 10 of *LIPIcs*, pages 71–80, 2011. doi: 10.4230/LIPIcs.RTA.2011.71.

[2] C. Sternagel and S. Winkler Certified Equational Reasoning via Ordered Completion In *Proc. 27th CADE*, 2019, to appear.

[3] S. Winkler and G. Moser MædMax: A Maximal Ordered Completion Tool In *Proc. 9th IJCAR*, volume 10900 of *LNCS*, pages 472–480, 2018. doi: 10.1007/978-3-319-94205-6_31.

*Supported by FWF (Austrian Science Fund) project T789.