From Compactifying Lambda-Letrec Terms to Recognizing Regular-Expression Processes

(Extended Abstract and Literature)

Clemens Grabmayer
Department of Computer Science
Gran Sasso Science Institute
L'Aquila, Italy

clemens.grabmayer@gssi.it

As a supplement to my talk at the workshop, this extended abstract motivates and summarizes my work with co-authors on problems in two separate areas: first, in the λ-calculus with letrec, a universal model of computation, and second, on Milner's process interpretation of regular expression, a proper subclass of the finite-state processes. The aim of my talk was to motivate a transferal of ideas for workable concepts of structure-constrained graphs from the problem of finding compact graph representations for terms in the λ-calculus with letrec to the problem of recognizing finite process graphs that can be expressed by regular expressions. In both cases the construction of structure-constrained graphs was expedient in order to enable to go back and forth easily between, on the one hand, terms and regular expressions, and on the other hand, term graphs and process graphs.

The main focus is on providing pointers to my work with co-authors, in both areas separately. A secondary focus is on explaining directions of my present projects, and describing research questions of possibly general interest that have developed out of my work in these two areas.

1 Introduction

The purpose of this extended abstract is to supplement my talk at the workshop [3] with a brief description of my work with co-authors in two areas, including ample references. While my workshop-presentation covered similar topics as my talk [1] at TERMGRAPH 2018, and while the proceedings article [2] for that workshop remains a useful resource, this article is a rewritten account with a detailed update on results that have been obtained in the meantime, and with an outlook on remaining challenging problems.

My talk [3] at the workshop aimed at motivating a fruitful transferal of ideas between two areas on which I worked in the (a bit removed, and more recent) past: λ-calculus, and the implementation of functional programming languages (2009–2014), and the process theory of finite-state processes (from 2016). My intention was to show, informally and supported by many pictures: How a solution to the problem of finding adequate graph representations for terms in the λ-calculus with letrec, a universal model of computation, turned out to be very helpful in understanding process graphs that can be expressed by regular expressions (via Milner's process interpretation), a proper subclass of finite-state processes.

In both cases the definition of an adequate notion of structure-constrained (term or process) graph was the key to solve a specific practical problem. It was central that the structure-constrained graphs facilitate to go back and forth easily between, on the one hand, terms in the λ-calculus with letrec and term graphs, and on the other hand, regular expressions and process graphs. The graph representations respect the appertaining operational semantics, but were conceived with specific purposes in mind: to optimize

Submitted to:
DCM 2023

functional programs in the Lambda Calculus with letrec; and respectively, to reason with process graphs denoted by regular expressions, and to decide recognizability of these graphs.

Section 2 summarizes work by Jan Rochel and myself that led us to the definition, and efficient implementation of maximal sharing for the higher-order terms in the λ-calculus with letrec. Specifically we formulated a representation-pipeline: Higher-order terms can be represented by, appropriately defined, higher-order term graphs, then these can be encoded as first-order term graphs, and subsequently those can in turn be represented as deterministic finite-state automata (DFAs). Via these correspondences and DFA minimization, maximal shared forms of higher-order terms can be computed.

Section 3 gives an overview of my work, in crucial parts done together with Wan Fokkink, on two non-trivial problems concerning the process semantics of regular expression. In Milner's process semantics, regular expressions are interpreted as nondeterministic finite-state automata (NFAs) whose equality is studied modulo bisimulation. Unlike for the standard language interpretation, not every NFA can be expressed by a regular expression. This raised a non-trivial recognition (or expression) problem, which was formulated by Milner (1984) next to a completeness problem for an equational proof system. In Section 3 I report on the crucial steps that have led me to a solution of the completeness problem.

Finally Section 4 reports on my present projects, and lists research questions that have developed out of my work in these two areas.

References

[1] Clemens Grabmayer (2018): *Modeling Terms by Graphs with Structure Constraints (Two Illustrations)*. Invited talk at the FSCD/FLoC Workshop *TERMGRAPH 2018*, Oxford, UK, July 7. Slides are available at `https://clegra.github.io/lf/TERMGRAPH-2018-invited-talk.pdf`.

[2] Clemens Grabmayer (2019): *Modeling Terms by Graphs with Structure Constraints (Two Illustrations)*. In Maribel Fernández & Ian Mackie, editors: *Proceedings Tenth International Workshop on Computing with Terms and Graphs, TERMGRAPH@FSCD 2018, Oxford, UK, 7th July 2018*, *EPTCS* 288, pp. 1–13, doi:10.4204/EPTCS.288.1.

[3] Clemens Grabmayer (2023): *From Compactifying Lambda-Letrec Terms to Recognizing Regular-Expression Processes*. Invited talk at the 13th International Workshop on *Developments in Computational Models*, affiliated with the conference FSCD, Sapienza Universita, Rome, July 2. Slides available at `https://clegra.github.io/lf/DCM-2023-invited-talk.pdf`.

2 Compactifying Lambda-Letrec Terms

This section summarizes work by Jan Rochel and myself in the framework of the NWO-project *Realizing Optimal Sharing (ROS)* at Utrecht University (2009–2014).[1] It eventually led us to the definition and practical implementation of maximal sharing for terms in the λ-calculus with letrec, a core language for the compilation of functional programming languages.

With the intention to study phenomena that arise practically for *optimal-sharing* implementations of the λ-calculus (by graph-transformation schemes by Lamping [18], and Kathail [17], and later interaction-net formalizations by Gonthier, Abadi, Lèvy [6], and also van Oostrom, van de Looij, Zwitserlood [20]), which are implementations of *optimal* or *parallel* β*-reduction* (due to Lèvy [19], Rochel wrote an impressive visualization and animation tool [21] for transforming graphs by reducing graph-rewrite redexes

[1] This project was headed jointly by Vincent van Oostrom (rewriting and λ-calculus) and Doitse Swierstra (implementation of functional languages). The project was concluded successfully in June 2016 with Jan Rochel's defense of his thesis [22].

per mouse-click. It produces beautifully rendered graphs that slowly float over the screen like bacteria in a liquid under a microscope. This animation tool provided us with much room for experimentation. When in trying to understand whether working with optimal implementations could avoid the so-called *static-argument transformation* we noticed that we could not establish that, we partly turned our attention away from optimal evaluation. We first described generalizations of the static-argument transformation that permit the avoidance of repetitive evaluation patters [23]. Realizing later that much of this optimization transformation was covered by *lambda dropping* [5], we set ourselves a more ambitious goal: to understand formally and conceptually the relationship between terms in the λ-calculus with letrec and the infinite λ-terms they represent.

1. Characterization of infinite unfoldings of programs ([11], RTA 2013),
2. Term graph representations ([13], TERMGRAPH 2013), and
3. Maximal Sharing for functional programs ([14], ICFP 2014).

Parameter-dropping optimization transformations In [23] we described an optimization transformation for the compilation of functional programs that drops parameters that are passed along unchanged between a number of recursive functions from the definitions of these functions. We formulated this optimization by means of higher-order rewrite rules.

Expressibility in $\lambda_{\mathsf{letrec}}$ and $\lambda\mu$ We obtained insightful characterizations [9], [11], [12] of those infinite λ-terms that arise as the unfolding semantics of terms in $\lambda_{\mathsf{letrec}}$, the λ-calculus with letrec, or of terms in the weaker system $\lambda\mu$, the λ-calculus with μ. Part of [9] is a non-trivial proof of confluence via the decreasing diagrams method of a higher-order rewriting system that defines the unfolding semantics of terms in $\lambda_{\mathsf{letrec}}$, also described in [24]. In [10] we showed confluence of let-floating operations on $\lambda_{\mathsf{letrec}}$-terms for a formalization of these operations as a higher-order rewriting system, in order to obtain unique normal form results for let-floating.

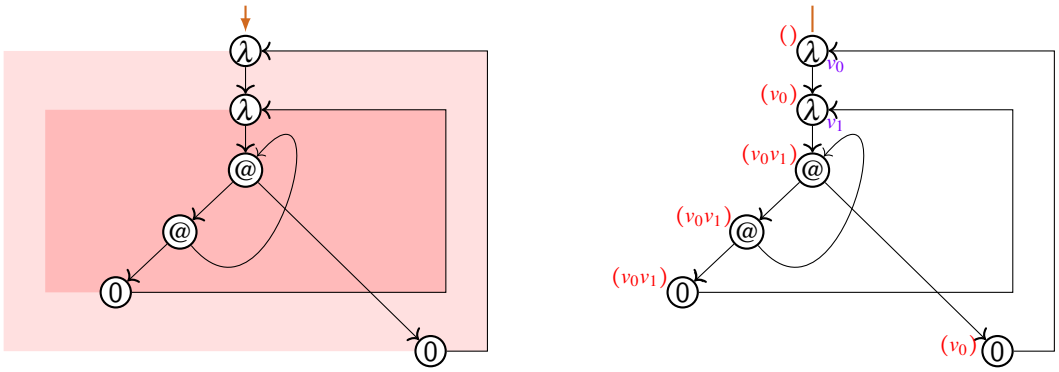
Term graph representations In [13], [16], we systematically investigated a range of natural options for representing cyclic λ-terms (like those represented by $\lambda_{\mathsf{letrec}}$-terms) by higher-order term graphs. As the result of this analysis we identified a natural class of higher-order term graphs that can be implemented faithfully as first-order term graphs.

Maximal sharing in $\lambda_{\mathsf{letrec}}$ We linked the language of $\lambda_{\mathsf{letrec}}$ to this class of higher-order term graphs via appropriate translation and readback operations. By utilizing our result on the implementation as term graphs, we obtained (see the ICFP-2014 article [14]) efficient methods for maximizing the degree of sharing that is expressed in a fixed $\lambda_{\mathsf{letrec}}$-term. This method also led to an efficient (basically quadratic-time) decision procedure for unfolding equivalence of $\lambda_{\mathsf{letrec}}$-terms. We implemented these methods by a Haskell tool [25] that is available on Haskell's Hackage platform.

▷ *Maximal sharing prototype and illustration tool [25]*:

Following the definition of maximally shared representations via a 'representation pipeline' in [14], this tool transforms a given functional program in the Core language of the λ-calculus with letrec into a term graph, and subsequently into a deterministic finite-state automaton (DFA). It prints intermediate representations, and graphically displays the obtained DFA. This DFA is then minimized, and finally a maximally shared representation of the original program is computed as the result.

Nested term graphs Motivated by the results on term graph representations and maximal sharing for $\lambda_{\mathsf{letrec}}$-terms, together with Vincent van Oostrom I developed a concept of nested term graph. It provides not only a natural formalization of the reasoning used in [13, 14], but it also makes this kind



λ-*higher-order term graph*, version 1: term graph with scope sets as structure constraints

λ-*higher-order term graph*, version 2: term graph with abstraction prefix function as structure constraint

Figure 1: Translation of the $\lambda_{\mathsf{letrec}}$-term $L_0 := \lambda x.\,\lambda f.\,\mathsf{let}\ r = f\,r\,x\ \mathsf{in}\ r$ into a λ-higher-order term graph with scope sets à la Blom (left), and a λ-h-o term graph $[\![L_0]\!]_{\mathcal{H}}$ with an abstraction-prefix function (right). Note that the inner scope has been chosen minimally here, applying eager scope closure.

reasoning much more broadly applicable, also outside of Lambda Calculus. We showed that nested term graphs can be implemented faithfully by first-order term graphs [8].

▷ *Maximal sharing prototype and illustration*:

Following the definition of maximally shared representations via a 'representation pipeline' in [14], this tool transforms a given functional program in the Core language of the λ-calculus with letrec into a term graph, and subsequently into a deterministic finite-state automaton (DFA). It prints intermediate representations, and graphically displays the obtained DFA. This DFA is then minimized, and finally a maximally shared representation of the original program is computed as the result.

▷ This tool [26] is available on Hackage via `http://hackage.haskell.org/package/maxsharing`.

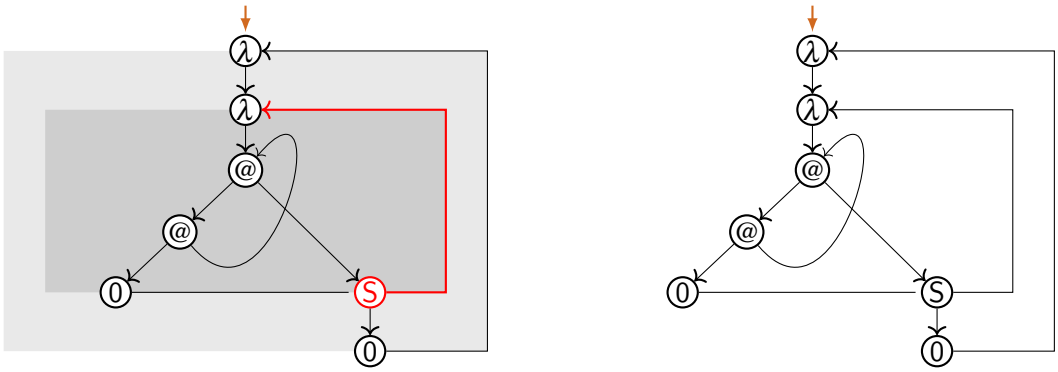
References

[4] Stefan Blom (2001): *Term Graph Rewriting, Syntax and Sematics*. Ph.D. thesis, Vrije Universiteit Amsterdam. Available at `https://ir.cwi.nl/pub/29853/29853D.pdf` from webpage `https://ir.cwi.nl/pub/29853` for this thesis at CWI Amsterdam.

[5] Olivier Danvy & Ulrik P. Schultz (2000): *Lambda-dropping: Transforming Recursive Equations into Programs with Block Structure*. *Theoretical Computer Science* 248(1), pp. 243–287, doi:https://doi.org/10.1016/S0304-3975(00)00054-2. Available at `https://www.sciencedirect.com/science/article/pii/S0304397500000542`. PEPM'97.

[6] Georges Gonthier, Martin Abadi & Jean-Jacques Lévy (1992): *The Geometry of Optimal Lambda Reduction*. In: *Proceedings of POPL'92*, pp. 15–26, doi:https://doi.org/10.1145/143165.143172.

[7] Clemens Grabmayer (2019): *Modeling Terms in the λ-Calculus with letrec*. Invited talk at the Workshop *Computational Logic and Applications*, Université de Versailles, France, July 1–2. Slides available at `https://clegra.github.io/lf/CLA-2019-invited-talk.pdf`.



first-order term graph with scope vertices with backlinks (+ scope sets)

λ-*term graph*: first-order term graph with scope vertices with backlinks

Figure 2: Translation of the $\lambda_{\mathsf{letrec}}$-term $L_0 := \lambda x.\lambda f.\mathsf{let}\ r = f\,r\,x\ \mathsf{in}\ r$ into a λ-term-graph $[\![L_0]\!]_{\mathcal{T}}$ by adding a scope vertex delimiting the inner scope to the λ-higher-order term graph in Fig. 1, and by then dropping the scope sets (which now can be reconstructed).

[8] Clemens Grabmayer & Vincent van Oostrom (2015): *Nested Term Graphs*. In Aart Middeldorp & Femke van Raamsdonk, editors: Post-Proceedings 8th International Workshop on *Computing with Terms and Graphs*, Vienna, Austria, July 13, 2014, *Electronic Proceedings in Theoretical Computer Science* 183, Open Publishing Association, pp. 48–65, doi:10.4204/EPTCS.183.4. ArXived at:1405.6380v2.

[9] Clemens Grabmayer & Jan Rochel (2012): *Expressibility in the Lambda-Calculus with Letrec*. Technical Report, `arxiv.org`, doi:10.48550/arXiv.1208.2383. arXiv:1208.2383.

[10] Clemens Grabmayer & Jan Rochel (2013): *Confluent Let-Floating*. In: *Proceedings of IWC 2013 (2nd International Workshop on Confluence)*, pp. 59–64. Available at `http://www.jaist.ac.jp/~hirokawa/iwc2013/iwc2013.pdf`. Available at `http://www.jaist.ac.jp/~hirokawa/iwc2013/iwc2013.pdf`.

[11] Clemens Grabmayer & Jan Rochel (2013): *Expressibility in the Lambda Calculus with Mu*. In Femke van Raamsdonk, editor: *24th International Conference on Rewriting Techniques and Applications (RTA 2013)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 21, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, pp. 206–222, doi:10.4230/LIPIcs.RTA.2013.206. Available at `http://drops.dagstuhl.de/opus/volltexte/2013/4063`.

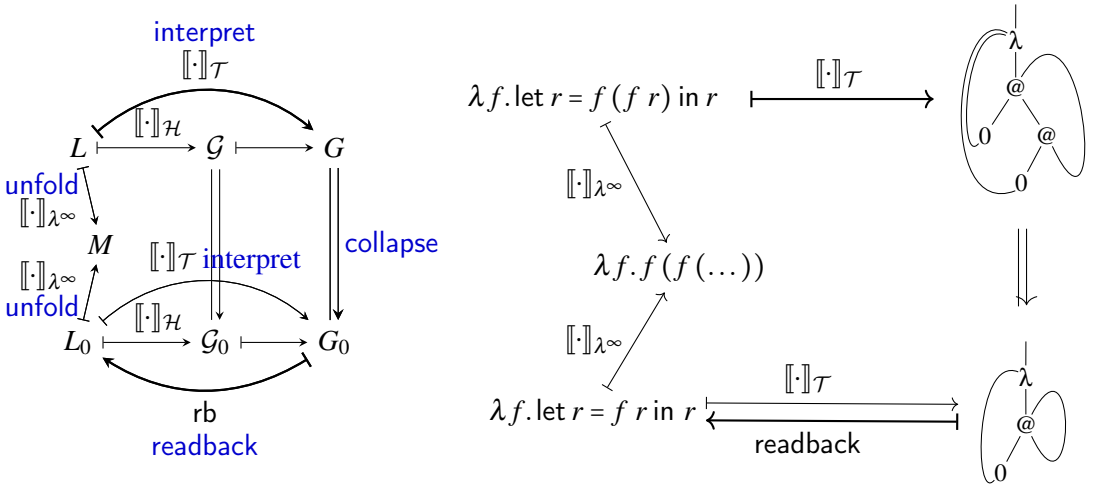
[12] Clemens Grabmayer & Jan Rochel (2013): *Expressibility in the Lambda Calculus with μ*. Technical Report, `arxiv.org`, doi:10.48550/arXiv.1304.6284. arXiv:1304.6284. Extends [11].

[13] Clemens Grabmayer & Jan Rochel (2013): *Term Graph Representations for Cyclic Lambda Terms*. In: *Proceedings of TERMGRAPH 2013*, *EPTCS* 110, pp. 56–73, doi:10.4204/EPTCS.110. ArXived at:1302.6338v1.

[14] Clemens Grabmayer & Jan Rochel (2014): *Maximal Sharing in the Lambda Calculus with Letrec*. In: *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, ACM, New York, NY, USA, pp. 67–80, doi:10.1145/2628136.2628148.

[15] Clemens Grabmayer & Jan Rochel (2014): *Maximal Sharing in the Lambda Calculus with letrec*. Technical Report, `arxiv.org`, doi:10.48550/arXiv.1401.1460. arXiv:1401.1460. Extends [14].

[16] Clemens Grabmayer & Jan Rochel (2014): *Term Graph Representations for Cyclic Lambda-Terms*. Technical Report, arxiv.org, doi:10.48550/arXiv.1304.6284. arXiv:1304.6284. Report extending [13] (proofs of main results added).



1. term graph interpretations $[\![\cdot]\!]$. of $\lambda_{\mathsf{letrec}}$-term L as:
   a. higher-order term graph $\mathcal{G} = [\![L]\!]_{\mathcal{H}}$
   b. first-order term graph $G = [\![L]\!]_{\mathcal{T}}$
2. bisimulation collapse $\Downarrow$ of first-order term graph G with as result G_0
3. readback rb of first-order term graph G_0 yielding $\lambda_{\mathsf{letrec}}$-term $L_0 = \mathsf{rb}(G_0)$.

Figure 3: Schematic representation of the maximal sharing method, and its application to a toy example: Maximum sharing proceeds of a $\lambda_{\mathsf{letrec}}$-term proceeds via three steps: interpretation as first-order term graphs, collapse of those via bisimilarity, and readback of $\lambda_{\mathsf{letrec}}$-terms from collapsed term graphs. On the top right these steps are illustrated for a redundant $\lambda_{\mathsf{letrec}}$-term formulation of a fixed-point combinator, yielding an efficient representation of such a combinator as $\lambda_{\mathsf{letrec}}$-term.

[17] Vinod Kumar Kathail (1990): *Optimal Interpreters for Lambda-calculus Based Functional Languages*. Ph.D. thesis, MIT. Available at `https://dspace.mit.edu/bitstream/handle/1721.1/14040/23292041-MIT.pdf?sequence=2`.

[18] John Lamping (1989): *An Algorithm for Optimal Lambda Calculus Reduction*. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, Association for Computing Machinery, New York, NY, USA, p. 16–30, doi:10.1145/96709.96711.

[19] Jean-Jacques Lévy (1978): *Réductions correctes et optimales dans le λ-calcul*. Ph.D. thesis, Université, Paris VII.

[20] Vincent van Oostrom, Kees-Jan van de Looij & Marijn Zwitserlood (2004): *]*. Extended Abstract for the Workshop on Algebra and Logic on Programming Systems (ALPS), Kyoto, April 10th 2004. Available at `http://www.phil.uu.nl/~oostrom/publication/pdf/lambdascope.pdf`.

[21] Jan Rochel (2010): *Port Graph Rewriting in Haskell*. Implementation with an explanatory technical report at `http://rochel.info/docs/graph-rewriting.pdf` tool on HackageDB with packages graph-rewriting, graph-rewriting-layout, graph-rewriting-gl, graph-rewriting-ski, graph-rewriting-trs, graph-rewriting-lambdascope, maxsharing. The rewrite system uses Stewart's port graph grammars (PGGs) [57].

[22] Jan Rochel (2016): *Unfolding Semantics of the Untyped λ-Calculus with* letrec. Ph.D. thesis, Utrecht University. Defended on June 20, 2016. Available at `http://rochel.info/thesis/thesis.pdf`.

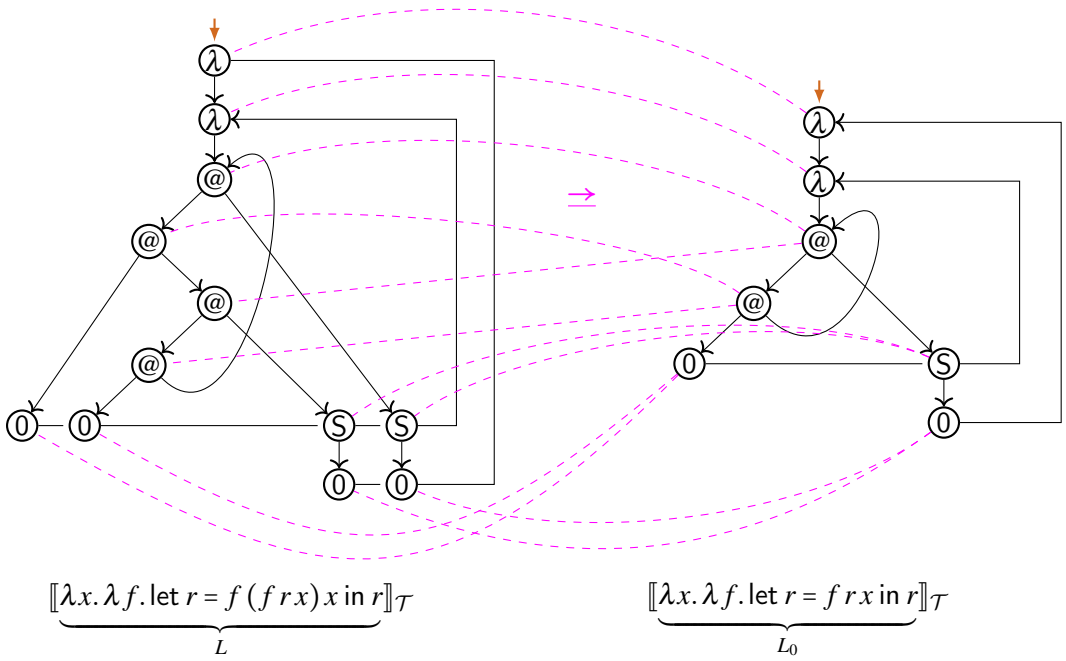


Figure 4: Compactification of the $\lambda_{\mathsf{letrec}}$-term L, a redundant form of a variant fixed-point combinator (compare with the forms in Figure 3), to the more compact $\lambda_{\mathsf{letrec}}$-term L_0. The λ-term-graph interpretations $[\![L]\!]_{\mathcal{T}}$ and $[\![L_0]\!]_{\mathcal{T}}$ of the $\lambda_{\mathsf{letrec}}$ terms L and L_0 are bisimilar. Indeed the links ---- form a functional bisimulation $\twoheadrightarrow$ from $[\![L]\!]_{\mathcal{T}}$ to $[\![L_0]\!]_{\mathcal{T}}$, of which the λ-term-graph $[\![L_0]\!]_{\mathcal{T}}$ is in bisimulation-collapsed form.

[23] Jan Rochel & Clemens Grabmayer (2011): *Repetitive Reduction Patterns in Lambda Calculus with Letrec*. In Rachid Echahed, editor: *Proceedings of the workshop TERMGRAPH 2011, 2 April 2011, Saarbrücken, Germany*, *Electronic Proceedings in Theoretical Computer Science* 48, Open Publishing Association, pp. 85–100, doi:10.4204/EPTCS.48.9. ArXived at:1102.2656v1.

[24] Jan Rochel & Clemens Grabmayer (2013): *Confluent Unfolding in the λ-calculus with* letrec. In: *Proceedings of IWC 2013 (2nd International Workshop on Confluence)*, pp. 17–22. Available at `http://www.jaist.ac.jp/~hirokawa/iwc2013/iwc2013.pdf`.

[25] Jan Rochel & Clemens Grabmayer (2014): *Maximal Sharing in the Lambda Calculus with letrec*. Implementation of the maximal sharing method described in [14, 15], available at `http://hackage.haskell.org/package/maxsharing/`.

[26] Jan Rochel & Clemens Grabmayer (2014): *Maximal Sharing in the Lambda Calculus with letrec*. Implementation of the maximal sharing method described in [14, 15], available at `http://hackage.haskell.org/package/maxsharing/`.

3 Proving Bisimilarity between Regular-Expression Processes

This section motivates, summarizes, and provides references to my work on Milner's process semantics of regular expressions [53]. An important part of it (leading to [50, 51]) was done in close collaboration with Wan Fokkink (2015–2020) who had stimulated me to work on Milner's question already in 2005. While this section focuses on my work on Milner's axiomatization questions (see **(A)** below), my current

$$\frac{}{1\Downarrow} \qquad \frac{e_i\Downarrow}{(e_1+e_2)\Downarrow} \qquad \frac{e_1\Downarrow \quad e_2\Downarrow}{(e_1\cdot e_2)\Downarrow} \qquad \frac{}{(e^*)\Downarrow}$$

$$\frac{}{a \xrightarrow{a} 1} \qquad \frac{e_i \xrightarrow{a} e_i'}{e_1+e_2 \xrightarrow{a} e_i'} \qquad \frac{e_1 \xrightarrow{a} e_1'}{e_1\cdot e_2 \xrightarrow{a} e_1'\cdot e_2} \qquad \frac{e_1\Downarrow \quad e_2 \xrightarrow{a} e_2'}{e_1\cdot e_2 \xrightarrow{a} e_2'} \qquad \frac{e \xrightarrow{a} e'}{e^* \xrightarrow{a} e'\cdot e^*}$$

Figure 5: Transition system specification $\mathcal{T}$ for computations enabled by regular expressions.

work on the expressibility question (see **(E)** below) will be mentioned in Section 4.

Milner introduced a process semantics $[\![\cdot]\!]_P$ in [53] for regular expressions (conceived by Kleene [52]) that refines the standard language semantics $[\![\cdot]\!]_L$ (defined by Copi, Elgot, Wright [33]). For regular expressions e that are constructed from constants 0, 1, letters over a given set A with the binary operators $+$ and $\cdot$, and the unary operator $(\cdot)^*$, Milner first defined a process interpretation $P(e)$ that can informally be described as follows: 0 is interpreted as a deadlocking process without any observable behavior, 1 as a process that terminates successfully immediately, letters from the set A stand for atomic actions that lead to successful termination; the binary operators $+$ and $\cdot$ are interpreted as the operations of choice and concatenation of two processes, respectively, and the unary star operator $(\cdot)^*$ is interpreted as the operation of unbounded iteration of a process, but with the option to terminate successfully before each iteration.

Milner formalized the process interpretation in [53] as process graphs that are defined by induction on the structure of regular expressions. But soon afterwards a formal definition by means of a transition system specification (TSS) that defines a labeled transition system (LTS) became more common. The TSS $\mathcal{T}$ in Figure 5 defines, via derivations that it permits from its axioms, labeled transitions $\xrightarrow{a}$ for actions a that occur in a regular expressions, and immediate successful termination via the unary predicate $\Downarrow$. The process interpretation $P(e)$ of a regular expression e is then defined as the sub-LTS that is induced by e in the LTS on regular expressions that is defined via derivability in $\mathcal{T}$. See Figure 6 for suggestive examples of (bisimilar) process interpretations of two simple regular expressions. In process graph illustrations there and later we indicate the start vertex by a brown arrow ➙, and the property of a vertex v to permit immediate successful termination by emphasizing v in brown as ◉ including an boldface ring. It is interesting to note that the so-defined process interpretation of regular expressions corresponds directly to non-deterministic finite-state automata (NFAs) that are defined via iterations of Antimirov's partial derivatives [28].[2]

Based on the process interpretation $P(\cdot)$, Milner then defined the process semantics of a regular expression e as $[\![e]\!]_P := [P(e)]_{\leftrightarrows}$ where $[P(e)]_{\leftrightarrows}$ is the equivalence class of $P(e)$ with respect to bisimilarity $\leftrightarrows$. In analogy to how language-semantics equality $=_{[\![\cdot]\!]_L}$ of regular expressions is defined from the language semantics $[\![\cdot]\!]_L$ (namely as $e =_{[\![\cdot]\!]_L} f$ if $L(e) = [\![e]\!]_L = [\![f]\!]_L = L(e)$, for all regular expressions e and f, where $L(g)$ is the language defined by a regular expression g) Milner was then interested in process-semantics equality $=_{[\![\cdot]\!]_P}$ that is defined, for all regular expressions e and f by:

$$\begin{aligned} e =_{[\![\cdot]\!]_P} f \;\; &:\Longleftrightarrow \;\; [\![e]\!]_P = [\![f]\!]_P \\ &\Longleftrightarrow \;\; P(e) \leftrightarrows P(f)\,. \end{aligned}$$

As the process interpretations of the regular expressions in Figure 7 are bisimilar, it follows that these regular expressions are linked by $=_{[\![\cdot]\!]_P}$.

[2] Antimirov did not have a process semantics in mind, but he had set out to define, for every regular expression e, an NFA

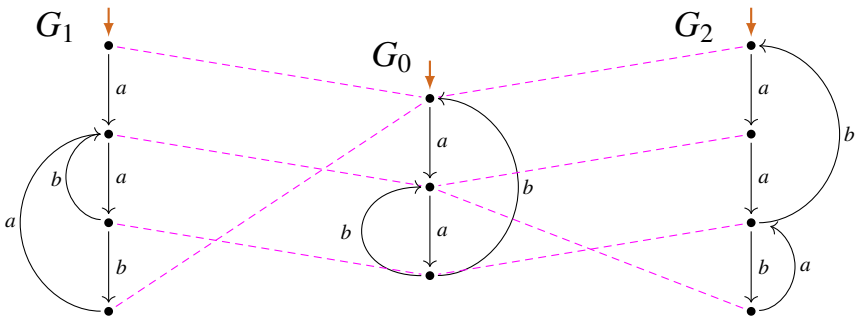


Figure 6: Two process graphs G_1 and G_2 that are P-expressible, and hence $[\![\cdot]\!]_P$-expressible, because they are the process interpretations of regular expressions as indicated. G_1 and G_2 are bisimilar via bisimulations that are drawn as links ---- to their joint bisimulation collapse G_0. It follows that also G_0 is $[\![\cdot]\!]_P$-expressible, and that process semantics equality holds between the regular expressions with interpretations G_1 and G_2, respectively. In this example G_0 is actually also in the image of $P(\cdot)$, hence P-expressible, as witnessed for example by $G_0 = P(((1\cdot a)\cdot(a\cdot(b+b\cdot a))^*)\cdot 0)$.

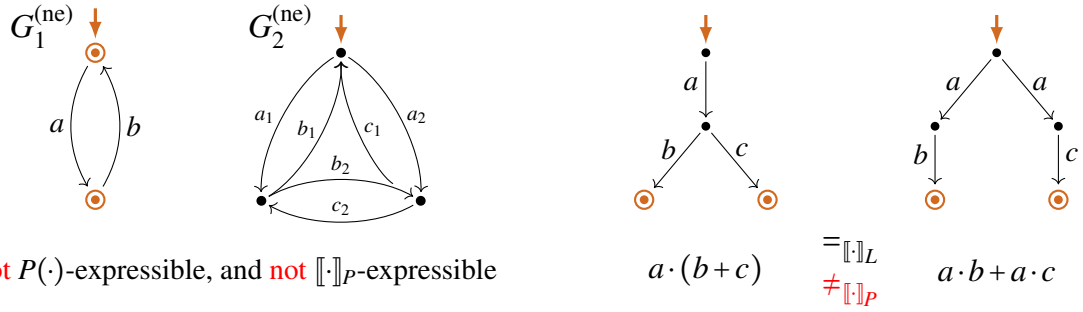


Figure 7: On the left: Two process graphs that are neither $P(\cdot)$-expressible (that is, not in the image of the process interpretation P) nor $[\![\cdot]\!]_P$-expressible (that is, not bisimilar to the process interpretation of any regular expression). On the right: two regular expressions with the same language semantics (associated language) but different process semantics, since the process interpretations are not bisimilar; therefore right-distributivity does not hold for $=_{[\![\cdot]\!]_P}$, which entails that fewer identities hold for $=_{[\![\cdot]\!]_P}$ than for $=_{[\![\cdot]\!]_L}$.

Milner realized in [53] that the process semantics $[\![\cdot]\!]_P$ of regular expressions differs from the language semantics $[\![\cdot]\!]_L$ in at least two respects: first, $[\![\cdot]\!]_P$ is incomplete, and second, process-semantics equality $=_{[\![\cdot]\!]_P}$ satisfies fewer identities than language-semantics equality $=_{[\![\cdot]\!]_L}$.

We start by explaining incompleteness of $[\![\cdot]\!]_P$. Language semantics $[\![\cdot]\!]_L$ is complete in the following sense: every language that can be accepted by a finite-state automaton (a regular language) is the language that is defined by a regular expression; that is every regular language is $[\![\cdot]\!]_L$-expressible. How-

that is typically smaller than the deterministic automaton (DFA) as usually associated with e in automata and language theory.

(A1) $e+(f+g)=(e+f)+g$
(A2) $e+0=e$
(A3) $e+f=f+e$
(A4) $e+e=e$
(A5) $e\cdot(f\cdot g)=(e\cdot f)\cdot g$
(A6) $(e+f)\cdot g=e\cdot g+f\cdot g$
(A7) $e=1\cdot e$
(A8) $e=e\cdot 1$
(A9) $0=0\cdot e$
(A10) $e^*=1+e\cdot e^*$
(A11) $e^*=(1+e)^*$

$$\frac{e=f\cdot e+g}{e=f^*\cdot g}\ \text{RSP}^*\ (\text{if } f\not\Downarrow)$$

Figure 8: Milner's equational proof system Mil for process semantics equality $=_{[\![\cdot]\!]_P}$ of regular expressions with the fixed-point rule RSP* in addition to the (not shown) basic rules for reasoning with equations (which guarantee that derivability in Mil is a congruence relation). From Mil the complete proof system for language equivalence $=_{[\![\cdot]\!]_L}$ due to Aanderaa arises by adding the axioms $e\cdot(f+g)=e\cdot f+e\cdot g$ and $e\cdot 0=0$ (which are not sound for $=_{[\![\cdot]\!]_P}$) and by dropping (A9) (which then is derivable).

ever, a comparable statement does not hold for the process interpretation. We call a finite process graph $[\![\cdot]\!]_P$-*expressible* if it is bisimilar to a *P-expressible* process graph, by which we the process interpretation of some regular expression (and hence a graph in the image of $P(\cdot)$. While it is easier to argue that not very finite process graph is *P*-expressible, there are also finite process graphs that are not $[\![\cdot]\!]_P$-expressible, neither. Milner proved in [53] that the process graph $G_2^{(\text{ne})}$ in Figure 7 not only is not *P*-expressible, but that it is also not $[\![\cdot]\!]_P$-expressible. He also conjectured that also $G_1^{(\text{ne})}$ in Figure 7 is not $[\![\cdot]\!]_P$-expressible; this was later shown by Bosscher [32].

Milner also noticed in [53] that some identities that hold for language-semantics equality $=_{[\![\cdot]\!]_L}$ are not true any longer for process semantics equality $=_{[\![\cdot]\!]_P}$. Most notably this is the case for right-distributivity $e\cdot(f+g)=e\cdot f+e\cdot g$, which is violated just as for the comparison of process terms via bisimilarity; see the well-known counterexample in Figure 7. The language-semantics identity $e\cdot 0=0$ is also violated in the process semantics. In order to define a natural sound adaptation (that we here designated by) Mil, see Figure 8, of the complete axiom systems for $=_{[\![\cdot]\!]_L}$ by Aanderaa [27] and Salomaa [54], Milner dropped these two identities from Aanderaa's system, but added the sound identity $0\cdot e=0$.

These two pecularities of the process semantics led Milner to formulating two questions concerning recognizability of expressible process graphs, and axiomatizability of process-semantics equality:

(E) How can $[\![\cdot]\!]_P$-expressible process graphs be characterized structurally, that is, those finite process graphs that are bisimilar to process interpretations of regular expressions?

(A) Is the natural adaptation Mil to process-semantics equality $=_{[\![\cdot]\!]_P}$ (see Figure 8 for Mil) of Salomaa's and Aanderaa's complete proof systems for language-semantics equality $=_{[\![\cdot]\!]_L}$ complete for $=_{[\![\cdot]\!]_P}$?

The expressibility question **(E)** seems to have received only limited attention at first. The reason may have been because it asks for a structural property of (the $[\![\cdot]\!]_P$-expressible) process graphs that is invariant under bisimilarity. This is a difficult aim, because bisimulations can significantly distort the topological structure of labeled transition graphs. Two variants of **(E)** have been solved after some time: First, the question for a natural sufficient condition for $[\![\cdot]\!]_P$-expressibility of process graphs was answered by Baeten and Corradini in [29] by the definition of process graphs that satisfy 'well-behaved' recursive specifications. Second, the question of whether $[\![\cdot]\!]_P$-expressibility of finite process graphs is decidable was answered by Baeten, Corradini, and myself in [30] by giving a decision procedure (unfortunately it is highly super-exponential) that is based on minimizing well-behaved specifications under bisimilarity.

For the axiomatization problem **(A)** at first only a string of partial results have been obtained. In particular Milner's proof system Mil has initially been shown to be complete for $=_{[\![\cdot]\!]_P}$ for the following subclasses of regular expressions:

(a) without 0 and 1, but with binary star iteration $e_1 {}^{\circledast} e_2$ with iteration-part e_1 and exit-part e_2 instead of unary star (Fokkink and Zantema, 1994, [37]),

(b) with 0, and with iterations restricted to exit-less ones $(\cdot)^* \cdot 0$ in absence of 1 (Fokkink, 1997, [36]) and in the presence of 1 (Fokkink, 1996 [35]),

(c) without 0, and with restricted occurrences of 1 (Corradini, De Nicola, and Labella, 2002 [34]),

(d) 1-*free* expressions formed with 0, without 1, but with binary iteration $\circledast$ (G, Fokkink, 2020, [50, 51], also showing the completeness of a proof system by Bergstra, Bethke, and Ponse [31]).

While the maximal subclasses in (c) and (d) are incomparable, these results can be joined to apply to an encompassing class that is still a proper subclass of the regular expressions, see [50]. Independently of these partial results concerning completeness of Milner's system Mil for subclasses of regular expressions, I noticed in [38] that from Mil a proof system that is complete for $=_{[\![\cdot]\!]_P}$ arises when the single-equation fixed-point rule RSP* is replaced by a unique-solvability principle USP for systems of guarded equations. Also in [38] I formulated a coinductively motivated proof system for process-semantics equality $=_{[\![\cdot]\!]_P}$ that utilizes Antimirov's partial derivatives [28] of regular expressions.

The principal new idea that facilitated the partial completeness result (d) in [50, 51] of Mil for 1-free regular expressions consisted in formulating a natural structural condition that is sufficient (but not necessary) for $[\![\cdot]\!]_P$-expressibility of process graphs: the *Loop Existence and Elimination Condition* LEE, and its layered form LLEE. This condition is based on the concept of 'loop (process) graph', and an elimination process of 'loop subgraphs' from a given process graph. A process graph G is said to have the property LEE if the non-deterministic iterative procedure, started on G, of repeatedly eliminating loop subgraphs is able to obtain a process graph without an infinite behavior (that is, a graph without infinite paths and traces). We explain the definitions in some more detail below, and provide examples.

A process graph LG is called a *loop (process) graph* if it satisfies the following three conditions:

(LG1) There is an infinite trace from the start vertex of LG.

(LG2) Every infinite trace from the start vertex v_s of LG returns to v_s.

(LG3) Immediate successful termination is only possible at the start vertex of LG.

In such a loop graph LG, the transitions from the start vertex are called *loop-entry transitions*, and all other transitions are called *loop-body transitions*. By a *loop subgraph* of a process graph G we mean a graph LG such that with respect to a vertex v of G, and a non-empty set T of transitions of G that depart from v the following three conditions are satisfied:

(LSG1) LG is a subgraph of G with start vertex v (which may be different from the start vertex v_s of G).

(LSG2) LG is generated by the transitions T from v in the following sense: LG contains all vertices and transitions of G that are reachable on traces that start from v via transitions in T, and continue onwards until v is reached again for the first time.

(LSG3) LG is a loop graph.

In accordance with the stipulation for loop graphs, in such a loop subgraph LG the transitions in T are called *loop-entry* transitions of LG, and all others *loop-body* transitions of LG. Based on these concepts, elimination of loop subgraphs is then defined as follows. We say that G' is the result of *eliminating a loop subgraph* LG with set T of loop-entry transitions *from* a process graph G, and denote such an elimination

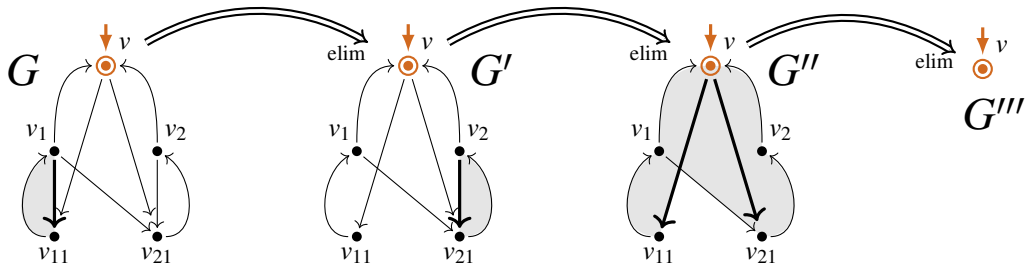


Figure 9: Example of successful loop elimination from the process graph G: three elimination steps of loop subcharts, which are represented as shaded gray areas, lead to the process graph G''' without infinite behaviour. These steps witness that G satisfies the properties LEE and LLEE (as well as do G', G'', G''').

step by $G \Rightarrow_{\text{elim}} G'$, if G' results from G by first removing the transitions in T and by then applying garbage collection of vertices and transitions that have become unreachable from the start vertex of G due to the transition removals. See Figure 9 for an example of three loop elimination steps. As for non-examples, note that neither of two not $[\![\cdot]\!]_P$-expressible graphs $G_1^{(\text{ne})}$ and $G_2^{(\text{ne})}$ in Figure 7 are loop graphs, nor do they contain loop subgraphs; hence neither of $G_1^{(\text{ne})}$ and $G_2^{(\text{ne})}$ permits a loop-elimination step.

We say that a process graph G *has the property* LEE (resp. *has the property* LLEE (*layered* LEE)) if there is a finite sequence of loop-elimination steps $G \Rightarrow^*_{\text{elim}} G'$ from G such that the resulting graph G' does not permit an infinite trace (and resp., if additionally during the elimination steps in $G \Rightarrow^*_{\text{elim}} G'$ it never happens that a transition is removed that was a loop-body transitions of a loop subgraph that was eliminated in an earlier step). It can be shown that although the property LLEE is a formally stronger requirement than the property LEE, which often helps to simplify proofs, both properties are equivalent. See Figure 9 for an example of a process graph G with the properties LEE and LLEE as is witnessed there by a sequence of three loop elimination steps that lead to graph G''' without infinite traces. The not $[\![\cdot]\!]_P$-expressible graphs $G_1^{(\text{ne})}$ and $G_2^{(\text{ne})}$ in Figure 7 do not satisfy LLEE and LEE, since loop elimination is not successful on them: they do not enable loop-elimination steps, but facilitate infinite traces.

The reason why the definition of the properties LEE and LLEE has facilitated progress concerning the problem **(A)** was that they define manageable conditions that could be used for proofs about process graphs that are linked by functional bisimulations. Specifically for obtaining the partial result (d) in [50, 51] it was crucial that we could prove the following facts:

(I)$_+$ Process interpretations of 1-free regular expressions satisfy LLEE (see [51, 50]).

(E)$_+$ Finite process graphs with LLEE are $[\![\cdot]\!]_P$-expressible, by 1-free regular expressions (see [51, 50]).

(C) LLEE is preserved along functional bisimilarity, and consequently, by the operation of bisimulation collapse (see [51, 50]).

Additionally, the property LLEE permitted me to formulate a coinductive version cMil of Milner's system Mil that also permits cyclic derivations of the form of process graphs with the property LLEE, see [41, 40, 48]. The system cMil could be viewed as being located proof-theoretically half-way in between Mil and bisimulations between process interpretation. As such it could be expected to form a natural beachhead for a completeness proof of Mil.

These results raised my hope that the argumentation could be extended quite directly to the full set of regular expressions with 1 and with unary iteration (instead of binary iteration) as well as to process graphs with 1-transitions and with the property LEE. While the generalization of **(I)$_+$** to all regular

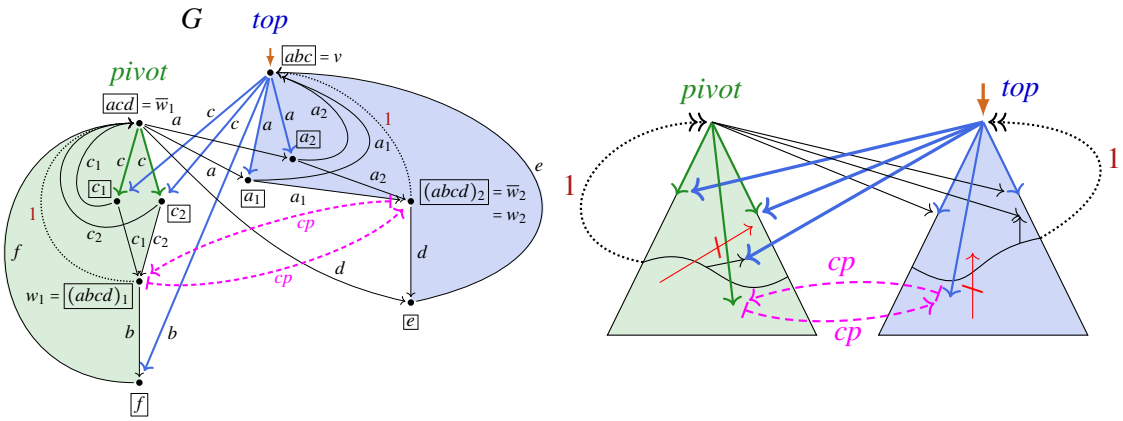


Figure 10: On the left: a finite process graph G with 1-transitions (drawn dotted, representing empty steps) that satisfies LLEE, but cannot be minimized under bisimilarity while preserving LLEE. It is a prototypical example of a twin-crystal. As such it consists of two interlinked parts, the top-part and the pivot-part, which by themselves are bisimulation collapsed, but contain vertices that have bisimilar counterparts in the opposite part of the twin-crystal. The self-inverse counterpart function *cp* links bisimilar vertices in the two parts. On the right: schematic illustration of a twin-crystal with suggestive drawing of its top-part and its pivot-part, together with interconnecting proper transitions from *top* and *pivot*.

expressions does not hold, I could show the existence of a refined process interpretation with the desired property, and generalize $\mathbf{(E)}_1$:

~~**(I)**~~ The process interpretation $P(e)$ of a regular expression e does not always satisfy LLEE (nor LEE) (see [39, 43]).

$\mathbf{(RI)}_1$ There is a refined process interpretation $\underline{P}(\cdot)$ that produces finite process graphs with 1-transitions such that, for every regular expression e, $\underline{P}(e)$ satisfies LLEE, $\underline{P}(e)$ is a refinement of $P(e)$ by sharing transitions by means of added 1-transitions, and $\underline{P}(e) \leftrightarrows P(e)$, that is, $\underline{P}(e)$ is bisimilar to $P(e)$ when 1-transitions are interpreted as empty steps (see [49], and a slightly weaker statement in [39, 43]).

$\mathbf{(E)}_1$ Finite process graphs with 1-transitions and with LLEE are $[\![\cdot]\!]_P$-expressible. (See [40, 41, 48]

However, critically, a direct generalization of our argument broke down dramatically due to the fact that the collapse statement **(C)** did not generalize to process graphs with LLEE that contain 1-transitions:

~~$\mathbf{(C)}_1$~~ LLEE is not preserved under bisimulation collapse of process graphs with 1-transitions. A counterexample holds for the process graph G on the left in Figure 10. (See also [45, 46].)

Consequently the proof strategy we used in [50, 51] for showing completeness of Mil for 1-free regular expressions, seemed not to work for showing completeness of Mil for the full class of regular expressions.

What came to my rescue here was that the counterexample for LLEE-preserving collapse of process graphs with 1-transitions and LLEE, the graph G in 10, is of a specific symmetric form that can be obtained by minimization under bisimilarity. It is a *twin-crystal*, a process graph with 1-transitions and with LLEE that is 'near-collapse in the sense that bisimilar vertices appear only as pairs. More precisely, twin-crystals are process graphs with 1-transitions and with LLEE that consist of a single strongly connected component (scc), and of two parts, the top-part and the pivot-part (see in Figure 10 on the right). Each part by itself is bisimulation collapsed, and then any two bisimilar vertices in the twin-crystal must occur in different of the top and pivot parts, and are linked by a self-inverse (partial) counterpart function. Process graphs with 1-transitions and with LLEE that are collapsed apart from

within scc's and in which all scc's are either collapsed or twin-crystals, we called *crystallized.* For this concept we could show:

(NC)$_1$ Every finite process graph with 1-transitions and with LLEE can be minimized under bisimilarity to a crystallized process graph (see [45, 46, 47].)

This statement is based on an effective *crystallization procedure* of process graphs with LLEE and with 1-transitions: it minimizes all scc's of the graph either to twin-crystals or collapsed parts of the graph, and also guarantees that the resulting graph is collapsed apart from within those scc's that are twin-crystals. The symmetric structure of twin-crystals can then be used to show that self-bisimulations of crystallized process graphs are of a particularly easy kind, which can be assembled from bisimulation slices that act on the twin-crystal-scc's [42]. This result on crystallized versions of process interpretations permitted me to adapt the proof strategy that Fokkink and I had used previously to also show completeness of Mil for $=_{[\![\cdot]\!]_P}$ on the full class of regular expressions, see [45, 46], and the poster [47].

There is much hope that the techniques we developed for solving the axiomatization question **(A)** will turn out to facilitate also significant improvements for answers to the expressibility question **(E)**. We will return to **(E)** at the end of the next section.

References

[27] Stål Aanderaa (1965): *On the Algebra of Regular Expressions*. Technical Report, Applied Mathematics, Harvard University.

[28] Valentin Antimirov (1996): *Partial Derivatives of Regular Expressions and Finite Automaton Constructions*. *Theoretical Computer Science* 155(2), pp. 291–319, doi:https://doi.org/10.1016/0304-3975(95)00182-4.

[29] Jos Baeten & Flavio Corradini (2005): *Regular Expressions in Process Algebra*. In: *Proceedings of LICS 2005*, IEEE Computer Society 2005, pp. 12–19, doi:10.1109/LICS.2005.43.

[30] Jos Baeten, Flavio Corradini & Clemens Grabmayer (2007): *A Characterization of Regular Expressions Under Bisimulation*. *Journal of the ACM* 54(2), pp. 1–28, doi:10.1145/1219092.1219094.

[31] Jan Bergstra, Inge Bethke & Alban Ponse (1994): *Process Algebra with Iteration and Nesting*. *The Computer Journal* 37(4), pp. 243–258, doi:10.1093/comjnl/37.4.243.

[32] Doeko Bosscher (1997): *Grammars Modulo Bisimulation*. Ph.D. thesis, University of Amsterdam.

[33] Irving M. Copi, Calvin C. Elgot & Jesse B. Wright (1958): *Realization of Events by Logical Nets*. *Journal of the ACM* 5(2), doi:10.1007/978-1-4613-8177-8_1.

[34] Flavio Corradini, Rocco De Nicola & Anna Labella (2002): *An Equational Axiomatization of Bisimulation over Regular Expressions*. *Journal of Logic and Computation* 12(2), pp. 301–320, doi:10.1093/logcom/12.2.301.

[35] Wan Fokkink (1996): *An Axiomatization for the Terminal Cycle*. Technical Report, *Logic Group Preprint Series*, Vol. 167, Utrecht University.

[36] Wan Fokkink (1997): *Axiomatizations for the Perpetual Loop in Process Algebra*. In: *Proc. ICALP'97*, *LNCS* 1256, Springer, Berlin, Heidelberg, pp. 571–581, doi:10.1007/3-540-63165-8_212.

[37] Wan Fokkink & Hans Zantema (1994): *Basic Process Algebra with Iteration: Completeness of its Equational Axioms*. *The Computer Journal* 37(4), pp. 259–267, doi:10.1093/comjnl/37.4.259.

[38] Clemens Grabmayer (2006): *A Coinductive Axiomatisation of Regular Expressions under Bisimulation*. Technical Report, University of Nottingham. Short Contribution to CMCS 2006, March 25-27, 2006, Vienna Institute of Technology, Austria, `https://clegra.github.io/lf/sc.pdf`, slides for the talk available at `https://clegra.github.io/lf/cmcs06.pdf`.

[39] Clemens Grabmayer (2020): *Structure-Constrained Process Graphs for the Process Semantics of Regular Expressions*. Technical Report, arxiv.org, doi:https://doi.org/10.48550/arXiv.2012.10869. arXiv:2012.10869. Report version of [43].

[40] Clemens Grabmayer (2021): *A Coinductive Version of Milner's Proof System for Regular Expressions Modulo Bisimilarity*. Technical Report, arxiv.org, doi:10.48550/arXiv.2108.13104. arXiv:2108.13104. Extended report for [41].

[41] Clemens Grabmayer (2021): *A Coinductive Version of Milner's Proof System for Regular Expressions Modulo Bisimilarity*. In Fabio Gadducci & Alexandra Silva, editors: *9th Conference on Algebra and Coalgebra in Computer Science (CALCO 2021)*, *Leibniz International Proceedings in Informatics (LIPIcs)* 211, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, pp. 16:1–16:23, doi:10.4230/LIPIcs.CALCO.2021.16. Extended report see [40].

[42] Clemens Grabmayer (2021): *Bisimulation Slices and Transfer Functions*. Technical report, Reykjavik University. Extended abstract for the 32nd Nordic Workshop on Programming Theory (NWPT 2021), http://icetcs.ru.is/nwpt21/abstracts/paper5.pdf.

[43] Clemens Grabmayer (2021): *Structure-Constrained Process Graphs for the Process Semantics of Regular Expressions*. *Electronic Proceedings in Theoretical Computer Science* 334, p. 29–45, doi:10.4204/eptcs.334.3. Extended report for [39].

[44] Clemens Grabmayer (2022): *A Coinductive Reformulation of Milner's Proof System for Regular Expressions Modulo Bisimilarity*. Technical Report, arxiv.org, doi:10.48550/arXiv.2203.09501. arXiv:2203.09501. Special-issue journal submission, whose development started from [41, 40].

[45] Clemens Grabmayer (2022): *Milner's Proof System for Regular Expressions Modulo Bisimilarity is Complete (Crystallization: Near-Collapsing Process Graph Interpretations of Regular Expressions)*. In: *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '22, Association for Computing Machinery, New York, NY, USA, pp. 1–13.

[46] Clemens Grabmayer (2022): *Milner's Proof System for Regular Expressions Modulo Bisimilarity is Complete (Crystallization: Near-Collapsing Process Graph Interpretations of Regular Expressions)*. Technical Report, arxiv.org, doi:10.48550/arXiv.2209.12188. arXiv:2209.12188. Technical report version of [45].

[47] Clemens Grabmayer (2022): *Milner's Proof System for Regular Expressions Modulo Bisimilarity is Complete (Crystallization: Near-Collapsing Process Graph Interpretations of Regular Expressions)*. Poster presented at LICS'22, Technion, Haifa, Israel, August 5. https://clegra.github.io/lf/poster-lics-2022.pdf.

[48] Clemens Grabmayer (2023): *A Coinductive Reformulation of Milner's Proof System for Regular Expressions Modulo Bisimilarity*. *Logical Methods in Computer Science* Volume 19, Issue 2, doi:10.46298/lmcs-19(2:17)2023. Available at https://lmcs.episciences.org/11519.

[49] Clemens Grabmayer (2023): *The Image of the Process Interpretation of Regular Expressions is Not Closed Under Bisimulation Collapse*. Technical Report, arxiv.org, doi:10.48550/arXiv.2303.08553. arXiv:2303.08553.

[50] Clemens Grabmayer & Wan Fokkink (2020): *A Complete Proof System for 1-Free Regular Expressions Modulo Bisimilarity*. In: *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '20, Association for Computing Machinery, New York, NY, USA, p. 465–478, doi:10.1145/3373718.3394744.

[51] Clemens Grabmayer & Wan Fokkink (2020): *A Complete Proof System for 1-Free Regular Expressions Modulo Bisimilarity*. Technical Report, arxiv.org, doi:10.48550/arXiv.2004.12740. arXiv:2004.12740. Report version of [50].

[52] Stephen C. Kleene (1951): *Representation of Events in Nerve Nets and Finite Automata*. In: *Automata Studies*, Princeton University Press, Princeton, New Jersey, USA, pp. 3–42, doi:10.1515/9781400882618-002.

[53] Robin Milner (1984): *A Complete Inference System for a Class of Regular Behaviours*. Journal of Computer and System Sciences 28(3), pp. 439–466, doi:10.1016/0022-0000(84)90023-0.

[54] Arto Salomaa (1966): *Two Complete Axiom Systems for the Algebra of Regular Events*. Journal of the ACM 13(1), pp. 158–169, doi:10.1145/321312.321326.

4 Current and Future Work

This section touches on my present research, and lists as well as briefly motivates some research questions and projects that have developed out of the work summarized in the past two sections. This is organized in two subsections that refer to the topics of Section 2 and Section 3, respectively.

4.1 Maximal Sharing at Run Time

Apart from using the maximal-sharing method for functional programs as a static-analysis based optimization transformation during compilation, one of the ideas for applications that Rochel and I gathered in [14] was that maximal sharing could be used as an optimization transformation also repeatedly at run-time. Making that idea fruitful, however, requires that representations of programs that are used in graph evaluators can be linked closely with λ-term-graph representations of $\lambda_{\mathsf{letrec}}$-terms on which the maximal-sharing method operates. This is necessary because graph evaluators in implementations of functional languages typically use supercombinator representations of $\lambda_{\mathsf{letrec}}$-terms, and much computational overhead is to be expected in transformations to and from λ-term-graphs. But any such overhead is highly undesirable during program execution. Yet supercombinator reduction as carried out by graph evaluators intuitively corresponds to *scope-sharing* forms of β-reduction.[3] But since λ-term-graphs contain neatly described scopes of λ-abstractions, the implementation of a scope-sharing form of evaluation on λ-term-graphs is conceivable. These considerations lead me to the following research question.

Research Question 1. *Coupling of maximal sharing with evaluation, generally, and more specifically:*

(i) Can the maximal-sharing method for terms in the λ-calculus with letrec *be coupled naturally with an efficient evaluation method (such as a standard graph-evaluation implementation)?*

(ii) Do λ-term-graphs (which represent $\lambda_{\mathsf{letrec}}$-terms) permit a representation as interaction nets or as port graphs [57] for which a form of β-reduction can be defined that is able to preserves, by adequately chosen multi-steps of interactions, scopes and also λ-term-graph form?

In communication after the workshop Ian Mackie pointed me to his interaction-net based implementation [55] of an evaluation method for the λ-calculus. I am grateful for this reference, first, because this interaction-net representation of λ-terms bears a close resemblance with λ-term-graphs, and second, because it provides a mechanism for implementing scope-preserving forms of β-reduction. Nevertheless it remains a challenging question to relate the two formalisms (λ-term-graphs and interaction-net representations of λ-terms in [55]) closely together. Yet an interaction-net representation of λ-term-graphs close to the representation of λ-terms as used in [55] seems to me to be a plausible and promising in-road for approaching part (ii) of Research Question 1.

[3] Note that scope-sharing is distinct from the *context-sharing* forms of graph reduction on which implementations of *parallel* or *optimal β-reduction* are based.

4.2 Crystallization: Proof Verification, and Application to the Expressibility Problem

Currently I am writing two articles that will provide the details of the completeness proof of Milner's proof system Mil. The first article is going to explain the motivation of the crystallization process for process interpretations of regular expressions: a limit to the minimization under bisimulation of process graphs that are expressible by a regular expression. This limit will be established specifically for the process graph G in Figure 10 with 1-transitions. The second article will detail the crystallization procedure by which process graphs with the property LEE (which are $[\![\cdot]\!]_P$-expressible) are minimized under bisimulation to obtain process graphs with LEE that are close to their bisimulation collapse. This central result will then be used, as explained in [45], to show that Milner's proof system Mil is complete with respect to process semantics equality $=_{[\![\cdot]\!]_P}$.

While the details of this completeness proof can be explained clearly conceptually, answering Milner's question **(A)** positively, a verification of the crystallization procedure and the completeness proof of Mil with respect to $=_{[\![\cdot]\!]_P}$ forms an important goal for me.

Research Project 2. *Formalization of the proofs for crystallization, and completeness of* Mil*:*

(a) Develop formalizations of structure constraints for process graphs in order to verify the correctness of the crystallization procedure for process graphs with LEE *by a proof assistant.*

(b) Use the correctness proof of crystallization to verify the completeness proof of Milner's proof system Mil *by a proof assistant.*

Separately I am working out the proof that the loop existence and elimination property LEE can be decided in polynomial time. For this result the observation is crucial that loop elimination $\Rightarrow_{\mathrm{elim}}$ can be completed to botain a confluent rewrite system (which is obviously terminating). As a consequence of the efficient decidability of LLEE it will follow that the restriction of the expressibility problem **(E)** to expressibility by regular expressions that are 1-*free under star* (but with unary iteration) can be solved efficiently. This is because the methods and results in [50, 51] permit to show that a finite process graph G is $[\![\cdot]\!]_P$-expressible by a regular expression that is 1-free under star if and only if the bisimulation collapse of G satisfies LLEE. Then it follows that expressibility of finite process graphs by regular expressions that are 1-free under star can be decided in polynomial time.

The crystallization procedure that we use in the completeness proof of Mil with respect to $=_{[\![\cdot]\!]_P}$ suggests that an extension of this characterization statement to one for $[\![\cdot]\!]_P$-expressibility in full generality is conceivable. We formulate that as our final research question.

Research Question 3. *Is the problem of whether a finite process graph is $[\![\cdot]\!]_P$-expressible efficiently decidable? That is, is there a polynomial decision algorithm for it? Or is $[\![\cdot]\!]_P$-expressibility at least fixed-parameter tractable (in* FPT*) for interesting parameterizations?*

References

[55] Ian Mackie (1998): *YALE: Yet Another Lambda Evaluator Based on Interaction Nets*. In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, Association for Computing Machinery, New York, NY, USA, p. 117–128, doi:10.1145/289423.289434.

[56] Ian Mackie (2004): *Efficient lambda evaluation with interaction nets*. In Vincent van Oostrom, editor: *Proceedings of RTA 2004, Aachen, Germany, June 3-5, 2004*, *LNCS* 3091, pp. 155–169, doi:10.1007/978-3-540-25979-4_11.

[57] Charles Stewart (2002): *Reducibility between Classes of Port Graph Grammar*. *Journal of Computer and System Sciences* 65(2), pp. 169 – 223, doi:http://dx.doi.org/10.1006/jcss.2002.1814.