


Dependency Pairs for Innermost Constrained Higher-Order Rewriting

Carsten Fuhs ✉ 

Birkbeck, University of London, United Kingdom

Liye Guo ✉ 

Radboud University, Nijmegen, The Netherlands

Cynthia Kop ✉ 

Radboud University, Nijmegen, The Netherlands

Abstract

Logically constrained simply-typed term rewriting systems (LCSTRSs) provide a form of rewriting geared towards analysis of programs with higher-order features and both algebraic and primitive data types. Termination of LCSTRSs has been studied for full rewriting without strategy assumptions. This extended abstract adapts the higher-order constrained Dependency Pair framework for innermost termination, which implies termination under call-by-value evaluation. The DP framework for innermost termination effectively handles universal computability. This provides a foundation for open-world termination analysis of programs using call-by-value evaluation via LCSTRSs.

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

Keywords and phrases Higher-order term rewriting, Logically constrained term rewriting, Innermost termination, Open-world termination, Call-by-value, Dependency pair framework.

Related Version A full version of this paper has been published at FSCD 2025 [5].

1 Introduction

In this extended abstract, we sketch techniques for proving *termination* of *Logically Constrained Simply-typed Term Rewriting Systems* (LCSTRSs) [14] for *call-by-value* evaluation. LCSTRSs are a higher-order functional intermediate verification language, designed as a compilation target for static analysis of programs with higher-order types. Our long-term goal is for LCSTRSs to be the cornerstone for a two-stage approach to static program analysis:

1. The *frontend* of the analysis: Given a program P written in (a practically relevant fragment of) the programming language L , translate P to an intermediate representation as an LCSTRS \mathcal{R}_P . This translation must (provably) preserve the properties of interest for our static analysis.
2. The *backend* of the analysis: Analyze \mathcal{R}_P using dedicated techniques for static analysis of LCSTRSs. The answer carries over to the original property of P .

This approach has been applied successfully across paradigms to multiple programming languages, such as Prolog [23, 10], Haskell [9], Scala [20], Java [21], and C [7], using various flavors of (constrained or unconstrained) term rewriting instead of LCSTRSs.

Classic term rewriting without any pre-defined data types and operations is known to be Turing-complete, which – in principle – makes it a sufficiently expressive intermediate language for this methodology. However, for *automated* static analysis, being able to represent programming language features directly, without cumbersome encodings, can make the difference between a program analysis tool that quickly finds a proof of the desired property and a tool that returns with an inconclusive result or times out.

We believe that LCSTRSs fill a “sweet spot” for such an intermediate representation. LCSTRSs integrate two strands of automated reasoning that have both proven useful for program analysis: term rewriting – here with support for higher-order functions – and Satisfiability Modulo Theories (SMT) solving.

In this extended abstract, we focus on step 2 of the above program analysis methodology. We analyze *termination* of the *call-by-value* rewrite relation and the (slightly more permissive) *innermost* rewrite relation of LCSTRSs. Call-by-value is a common evaluation strategy for many programming languages (e.g., OCaml, Scala, ...), so analyzing LCSTRSs directly for this rewrite strategy (or its generalization innermost rewriting) is a natural choice. Even for languages with *lazy* evaluation strategies, such as Haskell, past work [9] has shown how a frontend based on a form of abstract interpretation [3] can produce problems whose *innermost* termination implies termination of the lazy evaluation relation of the original program.

As is standard for termination analysis of rewriting, we build on *Dependency Pairs (DPs)* [1] and the *DP framework* [11, 12, 15]. The idea behind DPs is to prove termination of each function call separately. The DP framework allows us to combine different termination proving techniques, commonly referred to as *DP processors*, to simplify and decompose the termination proof obligation at hand. The DP framework was recently adapted to LCSTRSs [13]. However, this adaptation addresses termination for arbitrary rewrite strategies (“full termination”). While a termination proof in this setting is correct also for call-by-value evaluation, some proof methods that are sound for call-by-value or innermost evaluation are not applicable to full termination. Specifically, the analysis of *open-world* termination [13] – intuitively, termination of a set of rules also in the context of an arbitrary program known to be terminating on its own – is limited in power: The reduction pair processor [11, 12, 15] as the workhorse for termination proofs in the DP framework would be unsound.

In this extended abstract, we sketch an adaptation of the DP framework to proving call-by-value and innermost termination of LCSTRSs. We assume familiarity with term rewriting (see, e.g., [2]) and with the DP framework [11, 12, 15]. We use an informal presentation style for the introduced concepts, deliberately eliding some technical details. For formal definitions, theorems, proofs, as well as a discussion of experiments and related work, we refer to the full version of the paper published at FSCD 2025 [5].

2 Background

In this section, we recapitulate LCSTRSs [14] with the help of examples.

LCSTRSs are essentially an extension of (first-order) *Logically Constrained Term Rewriting Systems (LCTRSs)* [16] to a higher-order setting (without λ -abstractions). All variables and function symbols in an LCSTRS are *typed*, using curried simple types. The sorts underlying the type systems used for LCSTRSs represent either user-defined algebraic data types (as for many-sorted classic term rewriting) or pre-defined theory types such as integers or bitvectors taken from a logical theory (in the sense of SMT). The former correspond to user-defined data types in a programming language, and the latter represent primitive types as provided by many practically used programming languages. Types are then constructed inductively using a binary arrow constructor, which also allows for higher types for function arguments.

► **Example 1.** Consider the function declarations $\text{gcdlist} : \text{intlist} \rightarrow \text{int}$, $\text{fold} : (\text{int} \rightarrow \text{int} \rightarrow \text{int}) \rightarrow \text{int} \rightarrow \text{intlist} \rightarrow \text{int}$ and $\text{gcd} : \text{int} \rightarrow \text{int} \rightarrow \text{int}$. Here int is a theory type from integer arithmetic, whereas intlist is a user-defined data type with constructors $\text{nil} : \text{intlist}$ and $\text{cons} : \text{int} \rightarrow \text{intlist} \rightarrow \text{intlist}$. For our examples, we use integer arithmetic as the background

theory, so we include all values from \mathbb{Z} and the boolean values \mathbf{t} and \mathbf{f} as constructors, as well as standard arithmetic operations $(+, -, \dots)$, in the underlying signature of our LCSTRS.

We can now define the constrained rewrite rules of an LCSTRS \mathcal{R} to compute the greatest common divisor of a list of integers, where we omit constraints of the form $[\mathbf{t}]$:

$$\begin{array}{lll} \text{gcdlist} \rightarrow \text{fold gcd } 0 & \text{fold } f \ y \ \text{nil} \rightarrow y & \text{fold } f \ y \ (\text{cons } x \ l) \rightarrow f \ x \ (\text{fold } f \ y \ l) \\ \text{gcd } m \ n \rightarrow \text{gcd } (-m) \ n & [m < 0] & \text{gcd } m \ 0 \rightarrow m \quad [m \geq 0] \\ \text{gcd } m \ n \rightarrow \text{gcd } m \ (-n) & [n < 0] & \text{gcd } m \ n \rightarrow \text{gcd } n \ (m \bmod n) \quad [m \geq 0 \wedge n > 0] \end{array}$$

To see how an LCSTRS works, consider the following example evaluation, where the used redex is underlined: $\text{gcdlist} \ (\text{cons } (1 + 1) \ \text{nil}) \rightarrow_{\mathcal{R}} \text{fold gcd } 0 \ (\text{cons } (1 + 1) \ \text{nil}) \rightarrow_{\mathcal{R}} \text{gcd } (1 + 1) \ (\text{fold gcd } 0 \ \text{nil}) \rightarrow_{\mathcal{R}} \text{gcd } (1 + 1) \ 0 \rightarrow_{\mathcal{R}} \text{gcd } 2 \ 0 \rightarrow_{\mathcal{R}} 2$. The fourth step is a calculation step: $(1 + 1) \rightarrow_{\mathcal{R}} 2$. In the last step, $2 \geq 0$ holds, and we use the top-right gcd rule. Note that this evaluation is neither a call-by-value evaluation (“ \rightarrow_v ”: proper subterms of the used redex must be values; in our setting, constructor ground terms) nor an innermost evaluation (“ \rightarrow_i ”: proper subterms of the used redex must be in normal form). The reason is that in the second step, we did not rewrite the innermost redex $(1 + 1)$, which is not a value.

The following is an evaluation sequence that is both call-by-value and thus also innermost: $\text{gcdlist} \ (\text{cons } (1 + 1) \ \text{nil}) \xrightarrow{v}_{\mathcal{R}} \text{fold gcd } 0 \ (\text{cons } (1 + 1) \ \text{nil}) \xrightarrow{v}_{\mathcal{R}} \text{fold gcd } 0 \ (\text{cons } 2 \ \text{nil}) \xrightarrow{v}_{\mathcal{R}} \text{gcd } 2 \ (\text{fold gcd } 0 \ \text{nil}) \xrightarrow{v}_{\mathcal{R}} \text{gcd } 2 \ 0 \xrightarrow{v}_{\mathcal{R}} 2$. Note that in the first step, $(1 + 1)$ is *not* a subterm of the used redex.

► **Example 2.** To see the difference between call-by-value and innermost rewriting, consider the LCSTRS $\mathcal{R} = \{\text{hd} \ (\text{cons } x \ l) \rightarrow x, \text{tl} \ (\text{cons } x \ l) \rightarrow l\}$. Then $\text{hd} \ (\text{cons } 42 \ (\text{tl} \ \text{nil})) \xrightarrow{i}_{\mathcal{R}} 42$ is innermost but not call-by-value. The reason is that the subterm $\text{tl} \ \text{nil}$ of the redex is in normal form, but not a value: Innermost rewriting lets us rewrite above function calls that are “stuck” on their arguments; in OCaml, the computation would abort with an error.

► **Example 3** (Ex. 1 continued). Our goal is to prove that call-by-value evaluation with \mathcal{R} from Ex. 1 is terminating. However, termination of *innermost* rewriting, which is slightly more permissive than call-by-value evaluation, is more commonly considered in the term rewriting community and has been studied extensively for first-order term rewriting. Rather than reinvent the wheel, we formulate our techniques for innermost rewriting and provide a transformation to carry over (some) information about the intended call-by-value strategy.

The idea of the transformation is that in practice we may consider theory sorts to be *inextensible*: programmers are not allowed to add new constructors to pre-defined types like int . So we are interested only in instantiations of variables of inextensible theory types like int by their theory values. Now, for LCSTRSs, variables occurring in the constraint of a rule may be instantiated only by theory values during matching (and not, e.g., by normal forms that result from a function call “getting stuck”, analogous to Ex. 2). Thus, we add variables of inextensible theory sorts in rewrite rules to their constraints, writing φ, x_1, \dots, x_n for $\varphi \wedge x_1 \equiv x_1 \wedge \dots \wedge x_n \equiv x_n$. The operation \equiv corresponds to semantic equality in the underlying theory, so the only effect is to prevent undesired variable instantiation by non-values. For \mathcal{R} , the transformation adds variables to the constraints of four rules, where variables are added to a constraint only if they do not already occur there, e.g., $n < 0$ in the bottom-right rule already enforces that n may be instantiated only by values from \mathbb{Z} :

$$\begin{array}{lll} \text{fold } f \ y \ \text{nil} \rightarrow y & [\mathbf{t}, y] & \text{gcd } m \ n \rightarrow \text{gcd } (-m) \ n \quad [m < 0, n] \\ \text{fold } f \ y \ (\text{cons } x \ l) \rightarrow f \ x \ (\text{fold } f \ y \ l) & [\mathbf{t}, x, y] & \text{gcd } m \ n \rightarrow \text{gcd } m \ (-n) \quad [n < 0, m] \end{array}$$

We will refer to the result of the transformation as \mathcal{R}_{gcd} , and our goal will be to prove innermost termination of this LCSTRS.

Note that this approach of restricting innermost evaluation to call-by-value evaluation affects only variables of *theory* sorts like `int`. As terms of non-theory types like `intlist` are not allowed to occur in constraints of LCSTRS rules, the variable l in the bottom-left rule cannot be added to the constraint. This means that a term like `fold gcd 0 (cons 1 (tl nil))` can still be rewritten innermost using the bottom-left rule, even though this rewrite step is not call-by-value. Still, innermost termination of the LCSTRS produced by the transformation *implies* call-by-value termination of the input LCSTRS (but in general not vice versa).

3 Dependency Pairs for Innermost Termination of LCSTRSs

DPs [1] are a standard technique for proving termination of term rewriting systems. *Static DPs* (SDPs) [18, 19, 17, 6] are a common adaptation of DPs to the higher-order setting. While SDPs have soundness requirements based on computability (see, e.g., [6] for a discussion), SDPs tend to be applicable to LCSTRSs corresponding to real-world programs.

► **Example 4** (Ex. 3 cont'd). The set $\text{SDP}(\mathcal{R}_{\text{gcd}})$ of SDPs for \mathcal{R}_{gcd} consists of these SDPs:

- | | |
|--|---|
| (1) $\text{gcdlist}^\# l' \Rightarrow \text{gcd}^\# m' n'$ | (4) $\text{gcd}^\# m n \Rightarrow \text{gcd}^\# m (-n) [n < 0, m]$ |
| (2) $\text{gcdlist}^\# l' \Rightarrow \text{fold}^\# \text{gcd } 0 l'$ | (5) $\text{gcd}^\# m n \Rightarrow \text{gcd}^\# n (m \bmod n) [m \geq 0 \wedge n > 0]$ |
| (3) $\text{gcd}^\# m n \Rightarrow \text{gcd}^\# (-m) n [m < 0, n]$ | (6) $\text{fold}^\# f y (\text{cons } x l) \Rightarrow \text{fold}^\# f y l [t, x, y]$ |

As usual in the DP setting, “ $\#$ ” marks head symbols of potentially non-terminating function calls. Note that rules are implicitly flattened for the calculation of DPs so that variables are added as arguments until the rules have base type; hence the extra argument l' in the left-hand side of DP (1) and in both sides of DP (2). In contrast to the first-order setting, the right-hand side of a DP may contain variables that do not occur in the left-hand side, such as m' and n' in DP (1). The reason is that in the corresponding rewrite rule, the defined symbol `gcd` occurs in the right-hand side with fewer arguments than required for a step using a `gcd`-rule. At this point we do not know to which arguments `gcd` may eventually be applied. This is why we introduce fresh variables for the missing arguments, which in an innermost DP chain are instantiated by arbitrary (well-typed) normal forms. As we do not use SDPs to prove *non*-termination, correctness is not affected.

Our proof obligations are called *DP problems*, pairs $(\mathcal{P}, \mathcal{R})$ with \mathcal{P} a set of DPs and \mathcal{R} a set of rewrite rules. The initial DP problem for \mathcal{R}_{gcd} is $(\text{SDP}(\mathcal{R}_{\text{gcd}}), \mathcal{R}_{\text{gcd}})$. We must prove absence of infinite $(\mathcal{P}, \mathcal{R})$ -chains, analogously to the unconstrained first-order setting. For a DP problem (\emptyset, \mathcal{R}) , this holds trivially. As in the first-order innermost DP framework [11, 22], we keep track of the rewrite rules of the initial DP problem in a set \mathcal{Q} to have a faithful representation of the rewrite strategy (both components of a DP problem may be modified).

Proof techniques to simplify and decompose DP problems towards reaching DP problems of the shape (\emptyset, \mathcal{R}) are called *DP processors*. Many standard DP processors for (constrained and unconstrained) term rewriting can be adapted to the setting of full termination of LCSTRSs [13] and carry over to our innermost setting. These include the Dependency Graph processor, which decomposes the *dependency graph* of potentially consecutive DPs in a chain into its non-trivial strongly connected components (roughly: splits the call graph into mutually recursive calls, deletes other DPs), the subterm criterion and integer mappings (both delete DPs that make arguments smaller). These processors already suffice to prove innermost termination of \mathcal{R}_{gcd} (see [5] for details).

In the innermost setting, we can provide further processors that are practically relevant for program analysis. Rewrite systems that were obtained from an automatic translation by a frontend tend to contain many rewrite rules that make only small changes to the program

state – each instruction is translated separately. Thus, information that is needed for making progress in a termination proof may be spread over different rewrite rules or DPs.

► **Example 5.** Consider the below imperative program [7] (here in a Python-like syntax) on the left and the set \mathcal{P} of SDPs generated from this program on the right:

<code>def fact(x):</code>		$\text{fact}^\# x \Rightarrow u_1^\# x \ 1$	$[t, x]$
<code>z = 1</code>	# fact	$u_1^\# x \ z \Rightarrow u_2^\# x \ z \ 1$	$[t, x, z]$
<code>i = 1</code>	# u1	$u_2^\# x \ z \ i \Rightarrow u_3^\# x \ z \ i$	$[i \leq x, z]$
<code>while i <= x:</code>	# u2	$u_3^\# x \ z \ i \Rightarrow u_4^\# x \ (z * i) \ i$	$[t, x, z, i]$
<code>z = z * i</code>	# u3	$u_4^\# x \ z \ i \Rightarrow u_5^\# x \ z$	$[\neg(i \leq x), z]$
<code>i = i + 1</code>	# u4	$u_5^\# x \ z \ i \Rightarrow u_2^\# x \ z \ (i + 1)$	$[t, x, z, i]$
	# u5		

The only loop in the control-flow graph of the imperative program goes from program point u2 via u3 and u4 back to u2. Intuitively, the loop terminates because each time the instruction `i = i + 1` at program point u4 is executed, the value of `i` gets one step closer to the (constant) value of `x`, which is checked at program point u2. This means that the value of `x - i` decreases against the bound 0 in each iteration of the loop.

Unfortunately, the above set of DPs does not have the information “decrease” and “against a bound” together in a single DP. Processors like the integer mapping processor or standard reduction pair processors generally need to identify a decrease against a bound within the same DP. This is where *chaining processors* come to the rescue. They allow for merging consecutive DPs such that the resulting DPs carry out the operations of both original DPs.

In our example, chaining processors can iteratively remove the occurrences of $u_1^\#$, $u_3^\#$ and $u_4^\#$, and end with (1) $\text{fact}^\# x \Rightarrow u_2^\# x \ 1 \ 1 \ [t, x]$, (2) $u_2^\# x \ z \ i \Rightarrow u_2^\# x \ (z * i) \ (i + 1) \ [i \leq x, z]$, and (3) $u_2^\# x \ z \ i \Rightarrow u_5^\# x \ z \ [\neg(i \leq x), z]$. Now both the increase of `i` and the bound `i ≤ x` are represented in the single DP (2), and termination is easily proved using an integer mapping corresponding to our intuition from the imperative program.

Chaining processors occur in earlier work on constrained rewriting, e.g., for \mathcal{PA} -based TRSs [4] and for int-TRSs [8], which both correspond to constrained DPs with constraints over the integers and without nested function calls. Our chaining processor for the DP framework for LCSTRSs does not have the restriction on nested function calls.

Similar to chaining, also DP processors for *DP transformations* [1, 11, 12, 22] from the first-order DP framework modify the DPs themselves. Narrowing and rewriting require defined symbols from \mathcal{R} below the root of a right-hand side of a DP, so they would not be applicable in our example. A future adaptation of (forward) instantiation to the constrained setting *should* be able to propagate constraints between consecutive DPs, although it would not reduce the number of DPs like the chaining processor.

Further processors for the innermost DP framework for LCSTRSs include the usable rules processor, which deletes all rules from \mathcal{R} that are not called directly or indirectly from a DP, and the *reduction pair processor* using argument filterings for temporary removal of higher-order arguments to make reduction pair processors from the first-order world applicable (deletes DPs that make arguments smaller, can also handle arguments with function calls inside DPs). These processors are applicable even for compositional *open-world* termination analysis, where our LCSTRS is just a part of a larger (and growing) code base.

4 Conclusion

We have given an introduction to call-by-value and innermost evaluation using LCSTRSs, and we have sketched a framework for proving innermost termination of LCSTRSs using Dependency Pairs. For further details, please consider the full version of the paper [5].

References

- 1 T. Arts and J. Giesl. Termination of term rewriting using dependency pairs. *TCS*, 236(1–2):133–178, 2000. doi:10.1016/S0304-3975(99)00207-8.
- 2 F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.
- 3 P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In R. M. Graham, M. A. Harrison, and R. Sethi, editors, *Proc. POPL*, pages 238–252, 1977. doi:10.1145/512950.512973.
- 4 S. Falke and D. Kapur. A term rewriting approach to the automated termination analysis of imperative programs. In R. A. Schmidt, editor, *Proc. CADE*, pages 277–293, 2009. doi:10.1007/978-3-642-02959-2_22.
- 5 C. Fuhs, L. Guo, and C. Kop. An innermost DP framework for constrained higher-order rewriting. In M. Fernández, editor, *Proc. FSCD*, pages 20:1–20:24, 2025. doi:10.4230/LIPIcs.FSCD.2025.20.
- 6 C. Fuhs and C. Kop. A static higher-order dependency pair framework. In L. Caires, editor, *Proc. ESOP*, pages 752–782, 2019. doi:10.1007/978-3-030-17184-1_27.
- 7 C. Fuhs, C. Kop, and N. Nishida. Verifying procedural programs via constrained rewriting induction. *ACM TOCL*, 18(2):14:1–14:50, 2017. doi:10.1145/3060143.
- 8 J. Giesl, C. Aschermann, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, J. Hensel, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Analyzing program termination and complexity automatically with AProVE. *JAR*, 58(1):3–31, 2017. doi:10.1007/s10817-016-9388-y.
- 9 J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated termination proofs for Haskell by term rewriting. *ACM TOPLAS*, 33(2):7:1–7:39, 2011. doi:10.1145/1890028.1890030.
- 10 J. Giesl, T. Ströder, P. Schneider-Kamp, F. Emmes, and C. Fuhs. Symbolic evaluation graphs and term rewriting: a general methodology for analyzing logic programs. In A. King, editor, *Proc. PPDP*, pages 1–12, 2012. doi:10.1145/2370776.2370778.
- 11 J. Giesl, R. Thiemann, and P. Schneider-Kamp. The dependency pair framework: combining techniques for automated termination proofs. In F. Baader and A. Voronkov, editors, *Proc. LPAR*, pages 301–331, 2005. doi:10.1007/978-3-540-32275-7_21.
- 12 J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke. Mechanizing and improving dependency pairs. *JAR*, 37(3):155–203, 2006. doi:10.1007/s10817-006-9057-7.
- 13 L. Guo, K. Hagens, C. Kop, and D. Vale. Higher-order constrained dependency pairs for (universal) computability. In R. Kráľovič and A. Kučera, editors, *Proc. MFCS*, pages 57:1–57:15, 2024. doi:10.4230/LIPIcs.MFCS.2024.57.
- 14 L. Guo and C. Kop. Higher-order LCTRSs and their termination. In S. Weirich, editor, *Proc. ESOP*, pages 331–357, 2024. doi:10.1007/978-3-031-57267-8_13.
- 15 N. Hirokawa and A. Middeldorp. Automating the dependency pair method. *IC*, 199(1-2):172–199, 2005. doi:10.1016/J.IC.2004.10.004.
- 16 C. Kop and N. Nishida. Term rewriting with logical constraints. In P. Fontaine, C. Ringeisen, and R. A. Schmidt, editors, *Proc. FroCoS*, pages 343–358, 2013. doi:10.1007/978-3-642-40885-4_24.
- 17 K. Kusakari, Y. Isogai, M. Sakai, and F. Blanqui. Static dependency pair method based on strong computability for higher-order rewrite systems. *IEICE Trans. Inf. Syst.*, E92.D(10):2007–2015, 2009. doi:10.1587/transinf.E92.D.2007.
- 18 K. Kusakari and M. Sakai. Enhancing dependency pair method using strong computability in simply-typed term rewriting. *AAECC*, 18(5):407–431, 2007. doi:10.1007/s00200-007-0046-9.
- 19 K. Kusakari and M. Sakai. Static dependency pair method for simply-typed term rewriting and related techniques. *IEICE Trans. Inf. Syst.*, E92.D(2):235–247, 2009. doi:10.1587/transinf.E92.D.235.

- 20 D. Milovančević, C. Fuhs, and V. Kunčák. Proving termination of Scala programs by constrained term rewriting. In C. Kop and J. Voigtländer, editors, *Informal Proc. WPTE*, 2025. URL: <https://wpte2025.github.io/pre-proceedings.pdf>.
- 21 C. Otto, M. Brockschmidt, C. von Essen, and J. Giesl. Automated termination analysis of Java bytecode by term rewriting. In C. Lynch, editor, *Proc. RTA*, pages 259–276, 2010. doi:10.4230/LIPIcs.RTA.2010.259.
- 22 R. Thiemann. *The DP framework for proving termination of term rewriting*. PhD thesis, RWTH Aachen University, Germany, 2007. URL: <https://publications.rwth-aachen.de/record/62510>.
- 23 F. van Raamsdonk. Translating logic programs into conditional rewriting systems. In L. Naish, editor, *Proc. ICLP*, pages 168–182, 1997. doi:10.7551/mitpress/4299.003.0018.