# Runtime Complexity Analysis of Logically Constrained Rewriting

Sarah Winkler and Georg Moser

Free University of Bolzano, Italy, and University of Innsbruck, Austria
sarwinkler@unibz.it,georg.moser@uibk.ac.at

**Abstract.** Logically constrained rewrite systems (LCTRSs) are a versatile and efficient rewriting formalism that can be used to model programs from various programming paradigms, as well as simplification systems in compilers and SMT solvers. In this paper, we investigate techniques to analyse the worst-case runtime complexity of LCTRSs. For that, we exploit synergies between previously developed decomposition techniques for standard term rewriting by Avanzini *et al.* in conjunction with alternating time and size bound approximations for integer programs by Brockschmidt *et al.* and adapt these techniques suitably to LCTRSs. Furthermore, we provide novel modularization techniques to exploit loop bounds from recurrence equations which yield sublinear bounds. We have implemented the method in T$_{\mathsf{C}}$T to test the viability of our method.

## 1 Introduction

Rewriting with constraints over background theories is a highly versatile model of computation and tool for analysis. While user-defined data types are modelled by free function symbols, arbitrary decidable theories can be incorporated, such as integer or bit-vector arithmetic, lists, or array theory. Constraints over these theories can be effectively handled by SMT solvers. Different rewrite formalisms capture this idea [11,21,19]. Here we use the recent notion of *logically constrained term rewrite systems* (*LCTRSs* for short), due to Kop et al. [28,29,20,10].

LCTRSs can abstract programs in a variety of paradigms, comprising imperative, functional, and logic languages. They also subsume integer transition systems (ITSs), which constitute a frequently used program abstraction [19,15,8] but do—in contrast to LCTRSs—not support (non-tail) recursion. On the other hand, LCTRSs can also model simplification routines for expressions, which are crucial procedures in compilers or SMT solvers. For all of these application areas, LCTRSs offer a *uniform* toolset to analyse *termination* (or non-termination) [27,33], *reachability* [10], *uniqueness* [41], or *program equivalence* [20].

However, techniques for resource analysis of LCTRSs are so far lacking. This is despite the fact that in their application domains (program analysis, simplification systems), execution time is crucial. As a remedy, this paper investigates methods to analyse (worst-case) innermost runtime complexity of logically constrained rewrite systems. To this end, we unify and generalise the complexity framework for standard rewriting by Avanzini and Moser [4] with the approach

by Brockschmidt *et al.* to alternate time and size bound analysis for ITSs [8], and moreover propose processors for modularisation and sublinear bounds.

*Contributions.* We present a novel resource analysis framework for logically constrained rewrite systems (Sect. 4) coached in the modular processor framework of T$_\mathsf{C}$T [5]. Precisely,

1. we present the first fully-automated runtime complexity analysis of LTCRSs;
2. we unify the complexity framework for standard (innermost) rewriting by Avanzini and Moser [4] and the alternating time and size bound approximations for ITSs by Brockschmidt *et al.* [8],
3. generalising this, we introduce a novel modularisation processor, the *splitting processor*;
4. we present a novel processor, dubbed *recurrence processor* to derive sublinear bounds based on recurrences as described by the Master Theorem;
5. we illustrate the viability of our method by providing a prototype implementation as a dedicated module `tct-lctrs` in T$_\mathsf{C}$T, and evaluate it on ITS benchmarks.

In the remainder of the section, we highlight potential application areas of LCTRSs to emphasise their versatility. In the next section (Sect. 2) we give a high-level account of our technical achievements, providing a step-by-step explanation how the runtime complexity of a natural representation of mergesort can be optimally analysed in our framework. In this section, we also discuss to what extent our results can be applied to the below given examples. In Sect. 3 we summarise the foundations of LCTRSs, while in Sect. 4 we detail the complexity framework used. Processors carried over from the ITS setting are presented in Sect. 5, and the novel processors are introduced in Sect. 6. Implementational choices and experimental results are summarized in Sect. 7. Finally, in Sect. 8 we conclude. Some proofs were moved to an extended version [42].

*Logically Constrained Rewrite Systems.* We emphasise motivational examples from three different domains, focusing on imperative and logic programs, as well as compiler optimisations.

*Example 1.* The following recursive ITS $\mathcal{R}_1$, due to Albert et al. [1], corresponds to an imperative mergesort implementation after computing loop summaries. It is naturally coached into the LCTRS framework, with the theory of integers as background theory.

$$
\begin{array}{lll}
(1)\ \mathsf{init}(x,y,z) \to \mathsf{m}(x,y,z) & (2)\ \mathsf{m}_3(x,y,z) \to \mathsf{merge}(y,z,z) & \\
(3)\ \mathsf{m}_1(x,y,z) \to \mathsf{m}(y,y,z) & (4)\ \mathsf{merge}(x,y,z) \to \mathsf{merge}(x-1,y,z) & [x \geqslant 1 \wedge y \geqslant 1] \\
(5)\ \mathsf{m}_0(x,y,z) \to \mathsf{split}(x,y,z) & (6)\ \mathsf{split}(x,y,z) \to \mathsf{split}(x-2,y,z) & [x \geqslant 2] \\
(7)\ \mathsf{m}_2(x,y,z) \to \mathsf{m}(z,y,z) & (8)\ \mathsf{merge}(x,y,z) \to \mathsf{merge}(x,y-1,z) & [x \geqslant 1 \wedge y \geqslant 1] \\
(9)\ \mathsf{m}(x,y,z) \to \langle \mathsf{m}_0(x,u,v), \mathsf{m}_1(x,u,v), \mathsf{m}_2(x,u,v), \mathsf{m}_3(x,u,v) \rangle & & \\
\qquad [x \geqslant 2 \wedge u \geqslant 0 \wedge v \geqslant 0 \wedge x+1 \geqslant 2u \wedge 2u \geqslant x \wedge x \geqslant 2v \wedge 2v+1 \geqslant x] & &
\end{array}
$$

Here a rule of the form $\ell \to r\ [c]$ means that an instance of $\ell$ is replaced by the respective instance of $r$ provided that the instance of $c$ is satisfied.

Similarly, (constraint) logic programs can be nicely suited to LCTRSs.

*Example 2.* Consider the following simple Prolog program from the benchmarks collected by Mesnard and Neumerkl [31].

```
max_length(Ls,M,Len) :- max1(Ls,0,M), len(Ls,Len).
        len([H|T],L) :- len(T,LT), L is LT + 1.          len([],0).
     max1([H|T],N,M) :- H <= N, max1(T,N,M).             max1([],M,M).
     max1([H|T],N,M) :- H > N, max1(T,H,M).
```

Assuming an instantiated list `Ls`, `max_length(Ls,M,Len)` is deterministic and returns the maximal list entry and the length of the list. This function becomes representable as the following LCTRS $\mathcal{R}_2$ over the theory of integers and lists:

$$\begin{aligned}
\mathsf{max_length}(ls, m, l) &\to \langle \mathsf{max}(ls, 0, m), \mathsf{len}(ls, l)\rangle \\
\mathsf{len}(xs, l) &\to \mathsf{len}(t, l-1)\ [xs \approx h :: t] \qquad \mathsf{len}([], 0) \to \langle\rangle \\
\mathsf{max}(xs, n, m) &\to \mathsf{max}(t, n, m)\ [h \leqslant n \wedge xs \approx h :: t] \quad \mathsf{max}([], m, m) \to \langle\rangle \\
\mathsf{max}(xs, n, m) &\to \mathsf{max}(t, h, m)\ [h > n \wedge xs \approx h :: t]
\end{aligned}$$

Here, :: denotes the cons operator and $\langle\cdot,\cdot\rangle$, $\langle\rangle$ are additional constructor symbols to collect the recursive calls of a rule. Conceptually LCTRSs appear as a good fit to express *constraint* logic programs as well, making use of the fact that constraints are natively supported.

In order to emphasise that LCTRSs are not confined to static program analysis, we present a final example which is concerned with program optimisation.

*Example 3.* The Instcombine pass in the LLVM compilation suite performs *peephole optimisations* to simplify expressions in the intermediate representation. The current optimisation set contains over 1000 simplification rules to e.g. replace multiplications by shifts or perform bitwidth changes. About 500 of them have recently been translated into the domain-specific language Alive [30], and subsequently into LCTRSs [41], resulting in rules of the following shape:

$$\begin{aligned}
\mathsf{add}(x, x) &\to \mathsf{shift_left}(x, \#\mathsf{x1}) \\
\mathsf{add}(\mathsf{add}(\mathsf{xor}(\mathsf{or}(x, a), y), \#\mathsf{x1}), w) &\to \mathsf{sub}(w, \mathsf{and}(x, b))\ [a \approx \sim b] \\
\mathsf{add}(\mathsf{xor}(x, a), z) &\to \mathsf{sub}(a + z, x)\ [\mathsf{isPowerOf2}(a + \#\mathsf{x1}) \wedge \dots]\ .
\end{aligned}$$

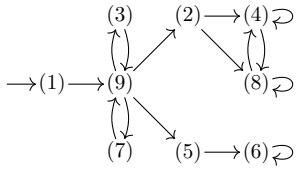
These rules are expressed over the background theory of bit-vectors. Naturally, as a compiler pass this simplification suite is a performance-critical routine, hence an automated complexity analysis is of great interest.

## 2 Step by Step to an Optimal Bound

Consider the rewrite system $\mathcal{R}_1$ from Ex. 1, and a rewrite sequence starting with an instance of $\mathsf{init}(x_0, y_0, z_0)$. Below we sketch the steps to obtain an upper bound on the runtime complexity of $\mathcal{R}_1$, expressed in $|x_0|$, $|y_0|$, and $|z_0|$, where $|\cdot|$ denotes the absolute value.

An automated runtime complexity analysis of mergesort is notoriously difficult: For this example, CoFloCo [17,16] can only derive a quadratic bound, while KoAT [8] (as well as AProVE [22]) even proposes an exponential bound. PUBS [1] can produce an $\mathcal{O}(n \cdot \log(n))$ bound, using a special *level-counting* feature, which however negatively affects its overall success rate. Due to the work presented in this paper, our complexity analyser T$_\mathsf{C}$T can automatically prove the optimal $\mathcal{O}(n \cdot \log(n))$ upper bound. This is obtained by the following recipe.

1. We first compute *dependency tuples* of all rules to focus the analysis on recursive calls (see Def. 4). Then a *dependency graph* approximation is computed to estimate computation paths, where the numbers refer to the respective dependency tuples of rules in Ex. 1:




2. Next, we derive bounds on the *size of variables* in left hand sides of rules, in terms of the sizes of the variables in the initial term $\mathsf{init}(x_0, y_0, z_0)$. For example, it is easy to check that for rule (9), $|x|$, $|y|$, and $|z|$ are bounded by $|x_0|$, $|y_0|$, and $|z_0|$, respectively, and all variables in other rules are bounded by $|x_0|$. This is established by the *size bounds processor* (Lem. 3). Formally, we adapt techniques developed for ITSs for that purpose [8].
3. We first derive time bounds for the SCCs $\{2, 4, 8\}$ and $\{6\}$ separately (Lem. 2). Thus, using the size bounds from above and suitable *interpretations* [4] (also called polynomial ranking functions [8]) for LCTRSs, one can derive linear runtime bounds $2|x_0| + 1$ and $|x_0|$ for these subproblems, respectively.
4. In order to analyse the SCC $\{3, 7, 9\}$, we first apply *chaining* to combine rule (9) with (3) and (7), respectively (eliminating symbols $\mathsf{m}_1$ and $\mathsf{m}_2$).
5. With respect to the modified rule (9) and the derived subproblem bounds, we exploit the *loop processor* (Lem. 5) to observe that its runtime can thus be overestimated by the following *recurrence* equations.

$$f(|x|, |y|, |z|) = 2 \cdot f(|x|/2, |x|/2, |x|/2) + 3|x| + 1 \quad f(1, |y|, |z|) = 0 \qquad (1)$$

Solving the recurrences by the Master Theorem, implies an overall runtime complexity of $\mathcal{O}(|x_0| \cdot \log(|x_0|))$ for $\mathcal{R}_1$, as $|x|$ in rule (9) is bound by $|x_0|$.

Wrt. $\mathcal{R}_2$ from Ex. 2, we can fully automatically infer an (asymptotic) optimal linear bound on the runtime complexity for the given instantiation. Here, we take an instance of $\mathsf{max\_length}(xs, z, l)$ as initial term. As for comparison, note that the corresponding logic program cannot be handled by a dedicated variant of AProVE [23] geared towards runtime complexity analysis of logic programs. Only termination can be shown by the most recent version of AProVE [22]. A priori, our approach is restricted to logic programs with instantiation patterns that

ensure determinism and avoid failure, but in the conclusion we discuss how to overcome this limitation.

Finally, Ex. 3 cannot yet be handled, as a successful analysis requires the extension of the proposed framework to *(innermost) derivational complexity* (i.e., the setting of arbitrary starting terms that may contain nested defined symbols). This is subject to future work. However, we conceive the work established in this paper as a solid first step towards the automated analysis of such systems.

## 3 Logically Constrained Term Rewriting

We assume familiarity with term rewriting [6,37], but briefly recapitulate the notion of logically constrained rewriting [28,20] that our approach is based on. We consider an infinite, sorted set of variables $\mathcal{V}$ and a sorted signature $\mathcal{F} = \mathcal{F}_T \uplus \mathcal{F}_L$ such that $\mathcal{T}(\mathcal{F}, \mathcal{V})$ denotes the set of terms over this disjoint signature. Symbols in $\mathcal{F}_T$ are called *term symbols*, while symbols in $\mathcal{F}_L$ are *theory symbols*. A term in $\mathcal{T}(\mathcal{F}_L, \mathcal{V})$ is a *theory term*. For a non-variable term $t = f(t_1, \dots, t_n)$, we write $\mathsf{root}(t)$ to obtain the top-most symbol $f$. A *position* $p$ is an integer sequence used to identify subterms, and the subterm of $t$ at position $p$ is denoted $t|_p$. We write $\mathcal{P}\mathsf{os}(t)$ for the set of positions in a term $t$, and given a set of function symbols $\mathcal{F}'$, $\mathcal{P}\mathsf{os}_{\mathcal{F}'}(t)$ are those positions $p \in \mathcal{P}\mathsf{os}(t)$ such that $t|_p$ is rooted by a symbol in $\mathcal{F}'$. A *substitution* $\sigma$ is a mapping from variables to terms with finite domain, and $t\sigma$ denotes the application of $\sigma$ to a term $t$.

Theory terms $\mathcal{T}(\mathcal{F}_L, \mathcal{V})$ have a fixed semantics: we assume a mapping $\mathcal{I}$ that assigns to every sort $\iota$ occurring in $\mathcal{F}_L$ a carrier set $\mathcal{I}(\iota)$. Moreover, we assume that for every element $a \in \mathcal{I}(\iota)$ there is exactly one constant symbol $c_a \in \mathcal{F}_L$, called a *value*. The set of all value symbols is denoted $\mathcal{V}\mathsf{al}$. For instance, if the sort of integers occurs in $\mathcal{F}_L$ then $\mathcal{V}\mathsf{al} \subseteq \mathcal{F}_L$ contains a value $c_i$ for every $i \in \mathbb{Z}$.

Moreover, we assume a fixed interpretation $\mathcal{J}$ that assigns to every theory symbol $f \in \mathcal{F}_L$ a function $f_\mathcal{J}$ of appropriate sort, and such that $(c_a)_\mathcal{J} = a$ for value symbols $c_a$, i.e., value symbols are interpreted as the represented element. The interpretation $\mathcal{J}$ naturally extends to theory terms without variables by setting $[f(t_1, \dots, t_n)]_\mathcal{J} = f_\mathcal{J}([t_1]_\mathcal{J}, \dots, [t_n]_\mathcal{J})$. In particular, we assume a sort $\mathsf{bool}$ such that $\mathcal{I}(\mathsf{bool}) = \{\top, \bot\}$ with values $\mathcal{V}\mathsf{al}_\mathsf{bool} = \{\mathsf{true}, \mathsf{false}\}$ such that $\mathsf{true}_\mathcal{J} = \top$, and $\mathsf{false}_\mathcal{J} = \bot$. We also assume that $\mathcal{F}_L$ contains equality symbols $\approx_\iota$ for every theory sort $\iota$, and a symbol $\wedge$ interpreted as logical conjunction.

Theory terms of sort $\mathsf{bool}$ are called *constraints*, and a *constrained term* is a pair $(t, \varphi)$ of a term $t$ and a constraint $\varphi$. A substitution $\gamma$ is a *valuation* if its range is a subset of $\mathcal{V}\mathsf{al}$. A constraint $\varphi$ is *valid*, denoted $\models \varphi$, if $[\varphi\gamma]_\mathcal{J} = \top$ for all valuations $\gamma$, and *satisfiable* if $[\varphi\gamma]_\mathcal{J} = \top$ for some valuation $\gamma$. We write $\psi \models \varphi$ if all valuations that satisfy $\psi$ also satisfy $\varphi$.

*Logically Constrained Rewriting.* A constrained rewrite rule is a triple $\ell \to r\ [\varphi]$ where $\ell, r \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, $\ell \notin \mathcal{V}$, $\varphi$ is a constraint, and $\mathsf{root}(\ell) \in \mathcal{F}_T$. If $\varphi = \mathsf{true}$ then the constraint is omitted. For a rule $\rho\colon \ell \to r\ [\varphi]$ we use $lhs(\rho) = \ell$ and $rhs(\rho) = r$ to denote its left- and right-hand sides, respectively. A set of constrained rewrite

rules is called a *logically constrained term rewrite system* (*LCTRS* for short). For an LCTRS $\mathcal{R}$, its *defined* symbols $\mathcal{F}_\mathcal{D}$ are all root symbols of left-hand sides, that is, $\mathcal{F}_\mathcal{D} = \{\mathsf{root}(\ell) \mid \ell \to r\ [\varphi] \in \mathcal{R}\}$. In the remainder we assume that LCTRSs are left-linear, that is, all variables occur at most once in the left-hand side $\ell$ of a rule $\ell \to r\ [\varphi]$.[1] An LCTRS $\mathcal{R}$ is a *transition system* if all rules in $\mathcal{R}$ are of the form $f(\ell_1, \ldots, \ell_n) \to g(r_1, \ldots, r_m)\ [\varphi]$ such that $f, g \in \mathcal{F}_T$, all $\ell_i \in \mathcal{V}$, and all $r_j$ are in $\mathcal{T}(\mathcal{F}_L, \mathcal{V})$; if moreover the background theory associated with $\mathcal{F}_L$ is the theory of integers then $\mathcal{R}$ is an *integer transition system* (ITS).

The fixed rewrite system $\mathcal{R}_{\mathsf{calc}}$ is the (infinite) set of rules $f(\ell_1, \ldots, \ell_n) \to u$ such that $f \in \mathcal{F}_L \setminus \mathcal{V}\mathsf{al}$, $\ell_i \in \mathcal{V}\mathsf{al}$ for all $1 \leqslant i \leqslant n$, and $u \in \mathcal{V}\mathsf{al}$ is the value symbol of $[f(\ell_1, \ldots, \ell_n)]_\mathcal{J}$. A rewrite step using $\mathcal{R}_{\mathsf{calc}}$ is called a *calculation step* and denoted $\to_{\mathsf{calc}}$. A *rule step* $s \to^\sigma_\rho t$ using a rule $\rho\colon \ell \to r\ [\varphi]$ and substitution $\sigma$ satisfies $s = C[\ell\sigma]$, $t = C[r\sigma]$, and $\sigma$ respects $\varphi$; where a substitution $\sigma$ is said to *respect* a constraint $\varphi$ if $\varphi\sigma$ is valid and $\sigma(x) \in \mathcal{V}\mathsf{al}$ for all $x \in \mathcal{V}ar(\varphi)$. The substitution in the notation $\to^\sigma_\rho$ is mostly omitted, and a rule step simply denoted $\to_\rho$. For an LCTRS $\mathcal{R}$, we denote the relation $\to_{\mathsf{calc}} \cup \{\to_\rho\}_{\rho\in\mathcal{R}}$ by $\to_\mathcal{R}$. The above rewrite step is *innermost*, denoted $s \overset{\mathsf{i}}{\to}_\rho t$, if all proper subterms of $\ell\sigma$ are in normal form with respect to $\to_\mathcal{R}$. Given binary relations $R$ and $S$, we write $R/S$ for $S^* \cdot R \cdot S^*$. For LCTRSs $\mathcal{R}$ and $\mathcal{S}$ we abbreviate $\overset{\mathsf{i}}{\to}_\mathcal{R} / \overset{\mathsf{i}}{\to}_\mathcal{S}$ by $\overset{\mathsf{i}}{\to}_{\mathcal{R}/\mathcal{S}}$, and $\overset{\mathsf{i}}{\to}_\mathcal{R} / \to_{\mathsf{calc}}$ by $\overset{\mathsf{i}}{\to}_{\mathcal{R}/\mathsf{calc}}$.

*Example 4 (continued from Ex. 2).* The LCTRS $\mathcal{R}_2$ indicated in Ex. 2, expressing the predicate `max_length`/3, makes use of the sorts $\mathsf{int}$, $\mathsf{list}$ and $\mathsf{bool}$. Furthermore, $\mathcal{F}_L$ consist of symbols :: and [] for lists, $\cdot$, $+$, $-$, $\leqslant$, and $\geqslant$ as well as values $n$ for all $n \in \mathbb{Z}$, with the usual interpretations on $\mathbb{Z}$ and lists of integers. Then $\mathcal{R}$ admits the following rewrite steps:

$$\mathsf{len}([1,2],2) \to \mathsf{len}([2],2-1) \to_{\mathsf{calc}} \mathsf{len}([2],1) \to \mathsf{len}([],1-1) \to_{\mathsf{calc}} \mathsf{len}([],0)$$

Note that in LCTRS rewriting, calculation steps like the subtractions in Ex. 4 are explicit in the $\to_{\mathsf{calc}}$ relation, in contrast to ITSs or related formalisms [32], where simplification is implicit. Moreover, innermost rewriting is a rather natural restriction for LCTRSs: By the definition of a rule step using some rule $\rho$, variables in the constraint of $\rho$ need to be substituted by values. Hence non-innermost steps are only possible if nested redexes occur below unconstrained variables. For instance, in a term $\mathsf{f}(\mathsf{f}(2))$ only the inner $\mathsf{f}$ call constitutes a redex for the rule $\mathsf{f}(x) \to x\ [x > 0]$.

*Algebras.* We assume mappings $|\cdot|_\iota : \mathcal{I}(\iota) \to \mathbb{N}$ for every sort $\iota$, playing the role of norms to measure size. For instance, one might take the absolute values for integers, the size function for arrays, and the unsigned integer value for bit-vectors. The subscript $\iota$ in $|t|_\iota$ is omitted if the sort of $t$ is clear from the context.

[1] Non-left-linear rules are rare in practice; and moreover repeated occurrences of a variable $x$ in $\ell$ can be substituted by a fresh variable $x'$, adding $x \approx x'$ to $\varphi$. Though this implies that $x$ can only be substituted by theory terms in rewrite sequences, for innermost evaluation this is not a limitation.

We consider well-founded algebras $\mathcal{A}$ over the natural numbers and the Booleans, with interpretation functions $f^{\mathcal{A}}$ for all $f \in \mathcal{F}_T \cup \mathcal{F}_L$, cf. [6,37]. By $t^{\mathcal{A}}$ we denote the interpretation of a term $t$ based on $\mathcal{A}$, and by $[\alpha]_{\mathcal{A}}(t)$ the interpretation of $t$ based on $\mathcal{A}$ and valuation $\alpha$. In order to bound complexity, we use algebras that incorporate the given complexity measures:

**Definition 1.** *A* measure interpretation *is given by an algebra $\mathcal{M}$ with carrier $\mathbb{N}$, and measures $|\cdot|_\iota$ for all sorts $\iota$. The interpretation $t^{\mathcal{M}}$ of a term $t$ is $|t|_\iota$ if $t \in \mathcal{V}$ has sort $\iota$, and $f^{\mathcal{M}}(t_1^{\mathcal{M}}, \dots, t_m^{\mathcal{M}})$ if $t = f(t_1, \dots, t_m)$. In addition, we demand that $f^{\mathcal{M}}([t_1]_{\mathcal{J}}^{\mathcal{M}}, \dots, [t_n]_{\mathcal{J}}^{\mathcal{M}}) \geqslant [f(t_1, \dots, t_n)]_{\mathcal{J}}^{\mathcal{M}}$ for all values $t_1, \dots, t_n$.*

In the following we suit interpretations (aka ranking functions) to LCTRSs. The ternary relation $>_{[\cdot]}^{\mathcal{M}}$ is defined as $s >_{[\varphi]}^{\mathcal{M}} t$ if and only if $[\alpha]_{\mathcal{M}}(s) > [\alpha]_{\mathcal{M}}(t)$ is satisfied for all valuations $\alpha$ that respect $\varphi$. Similarly, $s \geqslant_{[\varphi]}^{\mathcal{M}} t$ if and only if $[\alpha]_{\mathcal{M}}(s) \geqslant [\alpha]_{\mathcal{M}}(t)$ holds for all valuations $\alpha$ that respect $\varphi$.

**Definition 2.** *We call an LCTRS $\mathcal{R}$* weakly compatible *with a measure interpretation $\mathcal{M}$ if $\ell \geqslant_{[\varphi]}^{\mathcal{M}} r$ for all $\ell \to r\ [\varphi] \in \mathcal{R}$, and* strictly compatible *if $\mathcal{R}$ is weakly compatible and in addition $\ell >_{[\varphi]}^{\mathcal{M}} r$ for some $\ell \to r\ [\varphi] \in \mathcal{R}$.*

*Example 5.* Consider the measure interpretation $\mathcal{M}$ such that $\mathsf{m}_3^{\mathcal{M}}(x, y, z) = y$, $\mathsf{merge}^{\mathcal{M}}(x, y, z) = x$, $x +^{\mathcal{M}} y = x +_{\mathbb{N}} y$, $x -^{\mathcal{M}} y = \max(x -_{\mathbb{N}} y, 0)$, $\geqslant^{\mathcal{M}}$ is $\geqslant^{\mathbb{N}}$, and $v^{\mathcal{M}} = \max(v, 0)$ for all $v \in \mathbb{Z}$. The LCTRS $\mathcal{R}'$ consisting of the rules (2), (4), and (8) from Ex. 1 is strictly compatible with $\mathcal{M}$, since the rules (2) and (8) are weakly decreasing, while (4) is strictly decreasing.

## 4 Complexity Framework

An LCTRS $\mathcal{R}$ is *terminating* if $\to_{\mathcal{R}}$ is well-founded. In applications like static analysis, termination of a program is often not enough and more precise resource guarantees are needed. In this section we propose suitable runtime complexity notions for LCTRSs.

Following common notions in complexity analysis [4], the *derivation height* of a term $t$ wrt. a binary relation $\to$ is defined as follows: $\mathsf{dh}(t_0, \to) := \sup\{k \mid \exists\, t_1, \dots, t_k.\ t_0 \to \dots \to t_k\}$. We assume that an LCTRS $\mathcal{R}$ is associated with a unique *initial state* $(t_0, \varphi_0)$ such that $\varphi_0$ is a constraint and $t_0 = \mathsf{init}(\overline{x})$ is the *initial term*, for a vector of *input variables* $\overline{x} = (x_1, \dots, x_n)$ and a function symbol $\mathsf{init}$ that does not occur on any right-hand side. The intention is that we consider only rewrite sequences starting at $t_0\sigma$, such that $\sigma$ is a valuation that respects $\varphi_0$. Sometimes $s_0$ will be used as a shorthand for $(t_0, \varphi_0)$.

For $\overline{u}, \overline{v} \in \mathbb{N}^k$, let $\overline{u} \leqslant_k \overline{v}$ abbreviate $\bigwedge_{i=1}^{k} u_i \leqslant v_i$. Given $\overline{t} = (t_1, \dots, t_k)$, $|\overline{t}|$ denotes $(|t_1|, \dots, |t_k|)$, and $\overline{t}\sigma$ denotes $(t_1\sigma, \dots, t_k\sigma)$ for any substitution $\sigma$. For a term $t$, we write $\overline{\mathcal{V}ar}(t)$ for a vector containing $\mathcal{V}ar(t)$ in a fixed order.

**Definition 3.** *For an LCTRS $\mathcal{R}$ and a constrained term $(t, \varphi)$ such that $\overline{x} = \overline{\mathcal{V}ar}(t)$, the (innermost)* runtime complexity $\mathsf{rc}_{\mathcal{R}}^{(t,\varphi)} \colon \mathbb{N}^n \to \mathbb{N} \cup \{\omega\}$ *is defined as*

$$\mathsf{rc}_{\mathcal{R}}^{(t,\varphi)}(\overline{m}) = \sup\{\mathsf{dh}(t\sigma, \xrightarrow{\mathsf{i}}_{\mathcal{R}/\mathsf{calc}}) \mid\ |\overline{x}\sigma| \leqslant_n \overline{m} \text{ for some } \sigma \text{ that respects } \varphi\}.$$

Thus, the runtime complexity of an LCTRS is the maximal number of innermost *rule* steps in a rewrite sequence that starts with a size-bounded instance of the initial state $(t, \varphi)$; calculation steps are not counted. This is common in cost analysis, it also corresponds to the runtime complexity of a program or ITS [8], where the number of transitions are counted but not simplifications of expressions.

*Dependency pairs* are commonly used in termination and complexity analysis of rewrite systems. For termination of LCTRSs they were already used in earlier work [27]. For complexity analysis, stronger notions were developed for standard rewriting: *dependency tuples* (DTs) [34], *weak* [25], and *grouped* dependency pairs [4]. Since we consider innermost rewriting, we can use an LCTRS variant of dependency tuples. To that end, for every defined symbol $f$ we consider a fresh symbol $f^\sharp$, and for a term $t = f(t_1, \dots, t_n)$ write $t^\sharp$ to denote $f^\sharp(t_1, \dots, t_n)$.

**Definition 4.** *Consider a rule $\rho\colon \ell \to r\ [\varphi]$ such that $\mathcal{P}\mathsf{os}_{\mathcal{F}_\mathcal{D}}(r)$ is sorted as $p_1, \dots, p_k$ with respect to a fixed order on positions. Then the* dependency tuple $\mathsf{DT}(\rho)$ *of $\rho$ is the constrained rule $\ell^\sharp \to \langle (r|_{p_1})^\sharp, \dots, (r|_{p_k})^\sharp \rangle_k\ [\varphi]$. For an LCTRS $\mathcal{R}$, $\mathsf{DT}(\mathcal{R}) = \bigcup_{\rho \in \mathcal{R}} \mathsf{DT}(\rho)$.*

Here $\langle \dots \rangle_k$ is a fresh tuple symbol for every arity $k$ (but the subscript will be dropped for simplicity).

**Definition 5 (Dependency Graph).** *Let $\mathcal{R}$ be an LCTRS and $\mathcal{D} \subseteq \mathsf{DT}(\mathcal{R})$. The* dependency graph *(DG) is the directed graph with node set $\mathcal{D}$ and edges from $s^\# \to \langle t_1^\#, \dots, t_n^\# \rangle\ [\varphi]$ to $u^\# \to v\ [\psi]$ if there is some $t_i^\#$ such that $t_i^\# \sigma \to_\mathcal{R}^* u^\# \tau$, for some substitutions $\sigma$ and $\tau$ and some $i$, $1 \leqslant i \leqslant n$.*

The DG is not computable in general, but approximation techniques are well-known [27,34,5,22]. For instance, the graph in Sect. 2 constitutes a dependency graph approximation for the LCTRS from Ex. 1. Following Noschiniski *et al.* [34], we assume particular interpretation functions for the tuple operators $\langle \dots \rangle$. To this end, let a *DT-measure interpretation* $\mathcal{M}$ be a measure interpretation that interprets $\langle t_1, \dots, t_k \rangle^\mathcal{M} = t_1 + \dots + t_k$, for all $k \geqslant 0$.

Let the set of *bound expressions* UB be inductively defined as follows: (i) $|x|_\iota \in \mathrm{UB}$ for $x \in \mathcal{V}$ of sort $\iota$, (ii) $\mathbb{Z} \subseteq \mathrm{UB}$ and $\omega \in \mathrm{UB}$, (iii) if $p, q \in \mathrm{UB}$ then $p + q$, $pq$, and $max(p, q)$ are in UB, and (iv) if $p \in \mathrm{UB}$ and $k \in \mathbb{N}$ then $k^p$, $p/k$, and $\log_k(p)$ are in UB. Given $p, q \in \mathrm{UB}$, we write $p \leqslant q$ if $[\alpha]_\mathbb{N}(p) \leqslant [\alpha]_\mathbb{N}(q)$ for all substitutions $\alpha\colon \mathcal{V} \to \mathbb{N}$. For a bound expression $p \in \mathrm{UB}$ and $\overline{m} \in \mathbb{N}^n$ we also write $p(\overline{m})$ to denote the substituted bound expression $p[m_i/x_i]_{1 \leqslant i \leqslant n}$, assuming $\overline{x} \in \mathcal{V}^n$ are the variables in the initial term $t_0 = \mathsf{init}(\overline{x})$.

A triple $P = ((t, \varphi), \mathcal{D}, \mathcal{R})$ of a constrained term $(t, \varphi)$, a set of DTs $\mathcal{D}$, and an LCTRS $\mathcal{R}$ is called a *(complexity) problem*. Following Brockschmidt *et al.* [8], we next define time and size bound approximations.

**Definition 6.** *For a complexity problem $((t, \varphi), \mathcal{D}, \mathcal{R})$ with $\overline{x} = \overline{\mathcal{V}ar}(t)$, a function $T\colon \mathcal{D} \to \mathrm{UB}$ is a* runtime approximation *if, for all $\rho \in \mathcal{D}$ and $\overline{m} \in \mathbb{N}^n$,*

$$T(\rho)(\overline{m}) \geqslant \sup \{\mathsf{dh}(t\sigma, \xrightarrow{\mathsf{i}}_{\{\rho\}/\mathcal{D} \cup \mathcal{R}}) \mid |\overline{x}\sigma| \leqslant_n \overline{m} \text{ and } \sigma \text{ respects } \varphi\}.$$

In words, a runtime approximation $T(\rho)$ over-approximates how often a DT $\rho \in \mathcal{D}$ can be used in a rewrite sequence starting from the initial state, expressed in terms of the input variables. For instance, consider Ex. 1 and let $(1^\#)$ be the DT corresponding to rule (1). Then the function $T$ such that $T(1^\#) = 1$ and $T(\rho)(|x_0|, |y_0|, |z_0|) = |x_0|^2$ for all other DTs $\rho \in \mathcal{D}$ is a valid (though not optimal) runtime approximation.

For a complexity problem $((t, \varphi), \mathcal{D}, \mathcal{R})$, the set of *entry variables* EV is the set of all tuples $(\rho, y)$ such that $\rho \in \mathcal{D}$ and $y \in \mathcal{V}ar(lhs(\rho))$.

**Definition 7.** *For a complexity problem $((t, \varphi), \mathcal{D}, \mathcal{R})$ with $\overline{x} = \overline{\mathcal{V}ar}(t)$, a function $S\colon \mathrm{EV} \to \mathrm{UB}$ is a* size approximation *if*

$$S(\rho, y)(\overline{m}) \geqslant \sup\{|y\tau| \mid \exists \sigma,\ u.\ t\sigma \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}\cup\mathcal{D}} \cdot \xrightarrow{\mathsf{i}}{}^{\tau}_{\rho} u,\ |\overline{x}\sigma| \leqslant_n \overline{m}\}$$

*for $(\rho, y) \in \mathrm{EV}$ such that substitution $\sigma$ respects $\varphi$, and $\overline{m} \in \mathbb{N}^n$.*

A size approximation over-approximates how large a variable in the left-hand side of a rule in $\mathcal{D}$ can get in a rewrite sequence from the initial state, again expressed in terms of the input variables. A tuple $(T, S)$ is a *bound approximation* for a complexity problem $P$ if $T$ and $S$ are runtime and size approximations for $P$. We next define a complexity framework in the spirit of Avanzini and Moser [4].

**Definition 8.** *Given a complexity problem $P = (s_0, \mathcal{D}, \mathcal{R})$, a* (complexity) judgement *is a statement $\vdash P\colon (T, S)$, for functions $T : \mathcal{D} \to \mathrm{UB}$ and $S : \mathrm{EV} \to \mathrm{UB}$. The judgement is* valid *if $(T, S)$ is a bound approximation for $P$. A* complexity processor *is an inference rule on complexity judgements of the following form:*

$$\frac{\vdash P_1\colon (T_1, S_1), \ldots, \vdash P_k\colon (T_k, S_k)}{\vdash P\colon (T, S)}\ \mathsf{Proc}$$

*and it is* sound *if $\vdash P\colon (T, S)$ is valid whenever all $\vdash P_i\colon (T_i, S_i)$ are valid.*

For a problem $P = (s_0, \mathcal{D}, \mathcal{R})$ with initial state $s_0 = (\mathsf{init}(\overline{x}), \varphi)$, a DT $\ell \to r\ [\psi] \in \mathcal{D}$ is *initial* if $\mathsf{root}(\ell) = \mathsf{init}^\#$. The *initial processor* for $P$ is given by

$$\frac{}{\vdash P\colon (T, S_\omega)}\ \mathsf{Initial}$$

where $T(\rho) = 1$ if $\rho$ is initial and $T(\rho) = \omega$ otherwise; and $S_\omega(\rho, x) = \omega$ for all $(\rho, x) \in \mathrm{EV}$. Since $\mathsf{init}^\#$ does not occur on any right-hand side by assumption, the processor $\mathsf{Initial}$ is sound. For instance, the DT $\mathsf{init}^\#(x, y, z) \to \mathsf{m}^\#(x, y, z)$ originating from rule (1) in Ex. 1 is initial. For a problem $P = (s_0, \mathcal{D}, \mathcal{R})$ and an expression $C \in \mathrm{UB}$, we sometimes write $\vdash P\colon ((C)_\Sigma, S)$ to express that there is a runtime approximation $T$ such that $\vdash P\colon (T, S)$ and $C = \sum_{\rho\in\mathcal{D}} T(\rho)$.

The next result states that valid judgements bound the runtime complexity of LCTRSs. It can be proven in a similar way as [4, Theorem 6], using the properties of dependency tuples for innermost rewriting.

**Theorem 1.** *If an LCTRS $\mathcal{R}$ with initial state $(t, \varphi)$ admits the valid judgement $\vdash ((t^\#, \varphi), \mathsf{DT}(\mathcal{R}), \mathcal{R} \cup \mathcal{R}_{\mathsf{calc}})\colon (T, S)$ then $\mathsf{rc}^{(t,\varphi)}_{\mathcal{R}} \leqslant \sum_{\rho\in\mathsf{DT}(\mathcal{R})} T(\rho)$ holds.*

## 5 Processors

This section presents processors that implement the complexity framework from Sect. 4, in particular showing how the respective ITS techniques [8] carry over.

**Interpretation Processors.** Compatible interpretations are a standard tool in resource analysis, cf. [34,4,8]. We first present a processor using a measure interpretation that orients *all* rules and DTs (cf. [8, Theorem 3.6]). For $p \in \mathrm{UB}$, let $[p]$ denote the bound expression obtained from $p$ by replacing all coefficients in $p$ by their absolute values (such that the resulting expression is weakly monotone).

**Lemma 1.** *Let $P = ((t_0, \varphi_0), \mathcal{D}, \mathcal{R})$ and $\mathcal{M}$ a $\mathsf{DT}$-measure interpretation with which $\mathcal{R}$ is weakly, and $\mathcal{D}$ is strictly compatible. Then the following processor is sound, where $T'(\rho) = [(t_0)^{\mathcal{M}}]$ for all $\rho \in \mathcal{D}_>$, and $T'(\rho) = T(\rho)$ otherwise:*

$$\frac{\vdash P\colon (T, S)}{\vdash P\colon (T', S)} \quad \mathsf{Interpretation}$$

For instance, for Ex. 1 one can take the interpretation $\mathcal{M}$ such that $\mathsf{split}^{\mathcal{M}} = 0$ and $f^{\mathcal{M}} = 1$ for all other $f \in \mathcal{F}_T$, and symbols in $\mathcal{F}_L$ are interpreted as in Ex. 5. $\mathcal{R}_1$ is strictly compatible since all rules are weakly and rule (5) is strictly decreasing. This justifies a runtime approximation setting by $T(5^{\#}) = 1 = \mathsf{init}^{\#}(\overline{x})^{\mathcal{M}}$.

Next, we adapt [8, Theorem 3.6] to our setting, by which runtime bounds can be obtained using an interpretation that orients the given LCTRS *partially*. For a dependency graph $G$ and some $\mathcal{D}' \subseteq \mathcal{D}$, let $\mathsf{pre}(\mathcal{D}')$ be the set of all edges $(\rho_1, \rho_2)$ in $G$, such that $\rho_1 \in \mathcal{D} \setminus \mathcal{D}'$ and $\rho_2 \in \mathcal{D}'$. Moreover, for a DT $\rho$ with $\overline{\mathcal{V}ar}(lhs(\rho)) = (y_1, \ldots, y_k)$, let $\overline{S}_\rho$ denote $(S(\rho, y_1), \ldots, S(\rho, y_k))$.

**Lemma 2.** *Suppose $P = (s_0, \mathcal{D}, \mathcal{R})$ is a complexity problem such that $\mathcal{D}' \subseteq \mathcal{D}$ has no initial DTs, $\mathcal{R}$ is weakly, and $\mathcal{D}'$ is strictly compatible with a $\mathsf{DT}$-measure interpretation $\mathcal{M}$. Then the following processor is sound:*

$$\frac{\vdash P\colon (T, S)}{\vdash P\colon \left(\lambda\rho.\ \begin{cases} \sum_{(\gamma,\delta)\in\mathsf{pre}(\mathcal{D}')} T(\gamma) \cdot [lhs(\delta)^{\mathcal{M}}](\overline{S}_\delta) & \text{if } \rho \in \mathcal{D}'_> \\ T(\rho) & \text{otherwise} \end{cases}, S\right)} \quad \mathsf{TimeBounds}$$

*where $\mathcal{D}'_>$ is the set of rules $\ell \to r\ [\varphi]$ in $\mathcal{D}'$ such that $\ell >^{\mathcal{M}}_{[\varphi]} r$.*

Next, we define a proceessor to compute size approximations.

**Size Bounds.** Size approximations were developed for ITSs and tend to be less precise for LCTRSs due to nested terms. However, in many practical examples, a sufficient approximation is feasible. Next, we thus adapt the relevant notions to the LCTRS setting. First, the *local size approximation* overapproximates the size of entry variables in terms of variable sizes in predecessor rules.

**Definition 9.** *For $\delta, \rho \in \mathcal{D}$ and $(\rho, y) \in \mathrm{EV}$, let $S_{\delta\to\rho}\colon \mathcal{V} \to \mathrm{UB}$ be a* local size approximation *if*

$$S_{\delta\to\rho}(y)(\overline{m}) \geqslant \sup\{|y\tau| \mid \exists t, \sigma.\ \ell\sigma \to_\delta^\sigma \cdot \to_\rho^\tau t \text{ and } \overline{z}\sigma \leqslant_n \overline{m}\}$$

*where $\ell = lhs(\delta)$, $\overline{z} = \overline{\mathcal{V}ar}(\ell)$, and $\sigma$ is a valuation.*

The intention is that for an entry variable $(\rho, y)$, such that $y$ occurs in the left-hand side of $\rho$, the expression $S_{\delta\to\rho}(y)$ upper-bounds $y$ in terms of the variables in $\delta$, for the case where $\rho$ is applied after $\delta$. While such an expression is not always computable, it can often be over-approximated. For instance, in Ex. 1 a local size approximation $S_{(9)\to(2)}(y)$ could be $(|x|+1)/2$ or $|x|$: the subterm $\mathsf{m}_3^\#(x,u,v)$ on the right-hand side of (9) matches the left-hand side of (2), instantiating the variable $y$ by $u$, and the side condition of (9) ensures $x+1 \geqslant 2u$. We next define the *entry variable graph* to track the dependence of entry variables on each other. For $f \in \mathrm{UB}$, let $\mathcal{V}ar(f)$ be the set of all variables occurring in $f$.[2]

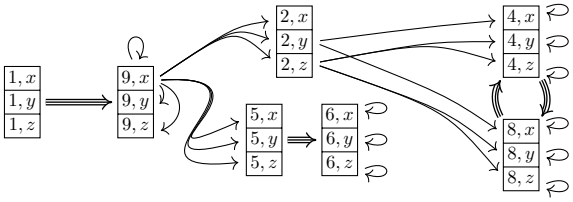
**Definition 10.** *An* entry variable graph $G_{\mathrm{EV}}$ *for $(s_0, \mathcal{D}, \mathcal{R})$ with DG $G$ has node set $\mathrm{EV}(\mathcal{D})$, and there is an edge from $(\delta, z)$ to $(\rho, y)$ labeled $S_{\delta\to\rho}(y)$ if $G$ has an edge from $\delta$ to $\rho$ and $z \in \mathcal{V}ar(S_{\delta\to\rho}(y))$.*

We illustrate the concept on our running example.

*Example 6.* Consider again Ex. 1. We first apply *chaining*, a standard technique in termination an complexity analysis [15,5], to compress the cycles $(9)-(3)-(9)$ and $(9)-(7)-(9)$ into single-step cycles, such that (9) is replaced by

$$\mathsf{m}(x,y,z) \to \langle \mathsf{m}_0(x,u,v), \mathsf{m}(u,u,v), \mathsf{m}(v,u,v), \mathsf{m}_3(x,u,v)\rangle\ [\psi]$$
$$\psi = x \geqslant 2 \wedge u \geqslant 0 \wedge v \geqslant 0 \wedge x+1 \geqslant 2u \wedge 2u \geqslant x \wedge x \geqslant 2v \wedge 2v+1 \geqslant x.$$

Then we obtain the following entry variable graph:




where a triple arrow $a \Longrightarrow b$ means that there are arrows from $(a,x)$ to $(b,x)$, $(a,y)$ to $(b,y)$, and $(a,z)$ to $(b,z)$. For all $(a,u) \in \mathrm{EV}$, all outgoing edges from $(a,u)$ can be labelled $|u|$, though more precise approximations are possible.

Next, we use the entry variable graph $G_{\mathrm{EV}}$ to obtain size bound refinements, following the approach of [8]. To that end, we define two processors $S_{triv}$ and $S_{scc}$ that refine bounds for trivial and non-trivial SCCs in $G_{\mathrm{EV}}$, respectively. Here, an SCC is *trivial* if it consists of a single node without an edge to itself.

[2] For more precision one could restrict to *active* variables, as done in [8].

**Definition 11.** *For size bounds $S$, we define $S_{triv}$ as follows: (i) $S_{triv}(\rho, y) = |y|$ if $\rho$ is initial; (ii) $S_{triv}(\rho, y) = \max\{\alpha(\overline{S}_\delta) \mid (\delta, z) \to^\alpha (\rho, y) \text{ in } G_{\mathrm{EV}}\}$, if $(\rho, y)$ is not in any non-trivial SCC of $G_{\mathrm{EV}}$; (iii) otherwise $S_{triv}(\rho, y) = S(\rho, y)$.*

We distinguish three types of edges in $G_{\mathrm{EV}}$, by partitioning their labels into the three sets $E_=$, $E_+$, and $E_\times$: for an edge labelled $\alpha$, (i) $\alpha \in E_=$ if $\alpha = a_\alpha \in \mathbb{N}$ or $\alpha = |x|$ for some $x \in \mathcal{V}$; (ii) $\alpha \in E_+$ if $|x| + a_\alpha \geqslant \alpha$ for some $x \in \mathcal{V}$ and $a_\alpha \in \mathbb{N}$; (iii) $\alpha \in E_\times$ if $c + \sum_{x \in \mathcal{X}} a_x |x| \geqslant \alpha$ for $c, a_x \in \mathbb{N}$ and $\mathcal{X} \subseteq \mathcal{V}$. For an SCC $C$ in $G_{\mathrm{EV}}$, let $C_\alpha$ denote the set of edge labels $\alpha$ of edges in $C$. For an entry variable graph $G_{\mathrm{EV}}$, let $\mathsf{pre}(\rho, y)$ be the set of all direct predecessors of $(\rho, y)$ in $G_{\mathrm{EV}}$.

**Definition 12.** *Let $(T, S)$ be a bound approximation and $C$ a non-trivial SCC in $G_{\mathrm{EV}}$. Then $S_{scc}$ is defined as (i) if $C_\alpha \subseteq E_=$ then $S_{scc}(\rho, y) = max\{\alpha \mid \alpha \in C_\alpha\}$, (ii) if $C_\alpha \subseteq E_+$ then let $\alpha_{pre} = max\{S(\rho', z) \mid (\rho', z) \in \mathsf{pre}(\rho, y) \setminus C\}$ and*

$$S_{scc}(\rho, y) = max(\{\alpha_{pre}\} \cup \{a_\alpha \mid \alpha \in C_\alpha\}) + \sum_{\rho \in \mathcal{D}} T(\rho) \cdot max\{a_\alpha \mid \alpha \in C \setminus E_=\}$$

*(iii) and $S_{scc}(\rho, y) = S(\rho, y)$ otherwise, for all $\rho \in C$ and $(\rho, y) \in \mathrm{EV}$.*

Both $S_{triv}$ and $S_{scc}$ are similar to the bounds developed in [8], though we omitted the case for $E_\times$ for reasons of space. We obtain soundness by similar proofs.

**Lemma 3.** *The following processors are sound:*

$$\frac{\vdash (s_0, \mathcal{D}, \mathcal{R}) \colon (T, S)}{\vdash (s_0, \mathcal{D}, \mathcal{R}) \colon (T, S_{triv})} \qquad \frac{\vdash (s_0, \mathcal{D}, \mathcal{R}) \colon (T, S)}{\vdash (s_0, \mathcal{D}, \mathcal{R}) \colon (T, S_{scc})} \quad \textsf{Size Bounds}$$

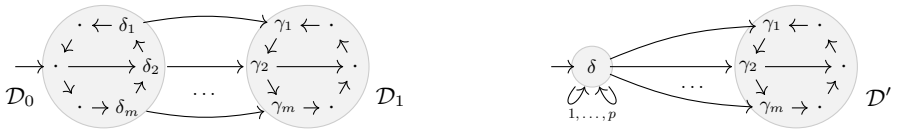
## 6 Processors for Splitting and Loop Summary

In this section we present new processors to decompose a problem into subproblems, as well as to analyse loops based on recurrence relations.

**Splitting.** We first consider a processor that allows to decompose a problem of a certain shape into two subproblems. To that end, let a subgraph be *forward closed* if it is closed under successors.

**Definition 13.** *Consider a problem $P = (s_0, \mathcal{D}, \mathcal{R})$ whose DG $G$ exhibits subgraphs $G_0$ and $G_1$ with node sets $\mathcal{D}_0$ and $\mathcal{D}_1$, respectively, such that $\mathcal{D} = \mathcal{D}_0 \uplus \mathcal{D}_1$, all initial DTs of $P$ are in $\mathcal{D}_0$, and $G_1$ is forward closed. Then $(\mathcal{D}_0, \mathcal{D}_1)$ is a* splitting *for $P$.*

A splitting thus decomposes a problem according to the scheme illustrated in Fig. 1a. The idea is that we first analyse the subproblems $P_0$ and $P_1$ corresponding to $\mathcal{D}_0$ and $\mathcal{D}_1$ separately, considering as initial states for $P_1$ all possible entry points $\gamma_i$. For DTs in $\mathcal{D}_0$ their time bounds in $P_0$ constitute overall time bounds since $G_1$ is forward closed; on the other hand, for every $\rho \in \mathcal{D}_1$, we compute time bounds via each entry point $\gamma_i$, and obtain an overall time bound by taking




(a) splitting (b) recurrence

Fig. 1: Problems of special shapes.

the sum over all $\gamma_i$. To that end, given $\gamma_i$, the time bound for $\rho$ in $P_1$ is applied to the size bound for $\gamma_i$, and multiplied by the time bound for the respective $\delta_i$, which upper-bounds the number of applications of $\delta_i$ followed by $\gamma_i$.

**Lemma 4.** *If $(s_0, \mathcal{D}, \mathcal{R})$ is a problem with splitting $(\mathcal{D}_0, \mathcal{D}_1)$ such that $\mathsf{pre}(\mathcal{D}_1) = \{(\delta_i, \gamma_i) \mid 1 \leqslant i \leqslant m\}$ and $\gamma_i = (\ell_i \to r_i\ [\varphi_i])$, the following processor is sound:*

$$\frac{\vdash P\colon (T,S) \quad \vdash (s_0,\mathcal{D}_0,\mathcal{R})\colon (T_0,S_0) \quad \bigwedge_{i=1}^m \vdash ((\ell_i,\varphi_i),\mathcal{D}_1,\mathcal{R})\colon (T_i,S_i)}{\vdash P\colon (\lambda\rho.\begin{cases} T_0(\rho) & \text{if } \rho \in \mathcal{D}_0 \\ \sum_{i=1}^m T_0(\delta_i)\cdot T_i(\rho)(\overline{S}_{\gamma_i}) & \text{if } \rho \in \mathcal{D}_1 \end{cases}, S)} \ \mathsf{Split}$$

Several improvements are conceivable, for instance the conditions of the initial states $(\ell_i, \varphi_i)$ could be strengthened using reachability analysis in the DG.

**Summarising Self-Loops.** We next propose a technique for the analysis of (sub)problems whose DG is of the shape shown in Fig. 1b. For vectors $\overline{a}, \overline{b}$, let $\overline{a} >_k \overline{b}$ be a shorthand for the expression $\overline{a} \geqslant_k \overline{b} \wedge (\bigvee_j a_j > b_j)$.

**Definition 14.** *Let $P = ((f(\overline{x}), \varphi), \mathcal{D}, \mathcal{R})$ with DG $G$ such that $\mathcal{D}$ can be written as $\mathcal{D} = \{\delta\} \uplus \mathcal{D}'$, the graph $G|_{\mathcal{D}'}$ is forward-closed in $G$, and $\delta$ is of the form:*

$$f(\overline{x}) \to \langle f(\overline{r}_1), \dots, f(\overline{r}_p), lhs(\gamma_1), \dots, lhs(\gamma_m)\rangle \quad [\psi] \tag{2}$$

*for $\{\gamma_1, \dots, \gamma_m\} \subseteq \mathcal{D}'$, such that $\overline{x}, \overline{r}_i \in \mathcal{T}(\mathcal{F}_L, \mathcal{V})^k$ and $\varphi \wedge \psi \models |\overline{x}| >_k |\overline{r}_i|$ for all $1 \leqslant i \leqslant p$. If there is moreover some $\overline{b} \in (\mathbb{N} \cup \{-\infty\})^k$ such that $\varphi \wedge \psi \models |\overline{x}| \geqslant_k \overline{b}$, then $P$ is* cyclic *with* termination condition $\overline{b}$.

**Lemma 5.** *Let $P = (s_0, \mathcal{D}, \mathcal{R})$ be a cyclic complexity problem with termination condition $\overline{b}$ and a DT $\delta$ of the form (2), and let $\gamma_i = (\ell_i \to r_i\ [\varphi_i])$, for all $i$, $1 \leqslant i \leqslant m$. Then the following processor is sound:*

$$\frac{\vdash P\colon (T,S) \quad \bigwedge_{i=1}^m \vdash ((\ell_i,\varphi_i),\mathcal{D}',\mathcal{R})\colon (T_i,S_i)}{\vdash (s_0,\mathcal{D},\mathcal{R})\colon (F(\overline{x})_\Sigma, S)} \ \mathsf{Recurrence}$$

*where $F$ is a solution to a recurrence $f(\overline{x}) = f(\overline{r}_1) + \ldots + f(\overline{r}_p) + H(\overline{x})$, $f(\overline{b}) = 0$ for some $H(\overline{x}) \geqslant \sum_{\rho \in \mathcal{D}'} \sum_{i=1}^m T_i(\rho)(\overline{S}_{\gamma_i})$.*

This processor is key to analyse the main loop in our running example.

*Example 7.* Consider Ex. 1 with chaining as applied in Ex. 6. For the subproblems $P_1 = (\mathsf{m}_0^\#(x,u,v),\psi), \mathcal{D}, \mathcal{R})$ and $P_2 = (\mathsf{m}_3^\#(x,u,v),\psi), \mathcal{D}, \mathcal{R})$ the judgements $\vdash P_1\colon ((x+1)_\Sigma, S)$ and $\vdash P_2\colon (u+v+1)_\Sigma, S)$ are valid, so we can set

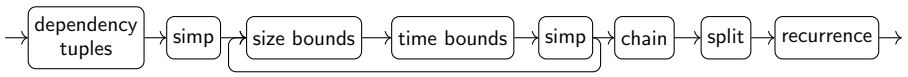
$H(x,u,v) = 2|x|+1 \geqslant x+u+v+1$ since $u, v \leqslant x/2$. Thus, we solve the recurrence (1) given in Sect. 2. According to one of the cases of the Master Theorem, (1) has a solution in $\mathcal{O}(|x| \cdot \log(|x|))$ which is a complexity approximation according to Lem. 5.

To simplify the presentation, we only considered cycles formed by a single DT, as indicated in Fig. 1b. The result generalizes to longer cycles, but chaining can often reduce these cases to the simpler situation discussed here.

## 7 Evaluation

To evaluate the viability of the presented framework, we prototyped our approach in the complexity analyser T$_\mathsf{C}$T [5].

*Implementation.* We added a new module `tct-lctrs` to the T$_\mathsf{C}$T tool suite, below we call the resulting tool T$_\mathsf{C}$T-LCTRS.[3] It currently supports the theory of integers, as well as some operations on lists. All processors described in this paper are implemented, using the modular processor framework of T$_\mathsf{C}$T. They are arranged in the following strategy, where the loop indicates exhaustive repetition:




We mention some implementation aspects that seem noteworthy.

- The simp processor combines some straightforward simplification processors: unsatisfiable paths, unreachable rules, and unused arguments are eliminated, and leaves in the DG obtain their time bound from their predecessors.
- Suitable algebras instantiating the interpretation and time bounds processors (Lems. 1 and 2) are searched for by means of an SMT encoding, as done in the ITS module of T$_\mathsf{C}$T previously using well-known techniques [35,7].
- Before applying the recurrence processor, T$_\mathsf{C}$T first applies chaining to obtain loops that involve only a single DT (see Appendix B for details).
- In the recurrence processor (Lem. 5),T$_\mathsf{C}$T first attempts to solve subproblems corresponding to the functions $h_1, \ldots, h_m$ separately, obtaining bound approximations $(T_i, S_i)$ for all $i$, $1 \leqslant i \leqslant m$ (see the notation of Lem. 5). Then, it is checked whether a function $H$ corresponding to one of the known recursion patterns satisfies $H(\overline{x}) \geqslant \sum_i \sum_{\rho \in \mathcal{D}'} T_i(\rho)$ using an SMT call.
- The splitting processor (Lem. 4) leaves a lot of choice to the implementation where to split. We currently use it to enable the loop processor, which requires a very particular problem shape.

If a subroutine requires an SMT query, T$_\mathsf{C}$T interfaces Yices [14] and Z3 [12].

*Experiments.* We evaluated T$_\mathsf{C}$T-LCTRS on the ITS benchmarks considered by Brockschmidt *et al.* [8], using a timeout of 60 seconds. Tab. 1 compares our implementation with KoAT [8], CoFloCo [17,16], the ITS version of T$_\mathsf{C}$T [5], and PUBS [1], giving the number of problems for which a bound was derived at all,

[3] The code is available from https://github.com/bytekid/tct-lctrs.

the number of constant bounds, and the number of bounds that are at most linear, quadratic, and cubic, respectively. The new splitting and recurrence pro-

| | TCT-LCTRS | KoAT | CoFloCo | TCT-ITS | PUBS |
|---|---|---|---|---|---|
| solved problems | 359 | 404 | 347 | 309 | 285 |
| constant | 119 | 131 | 117 | 118 | 109 |
| $\leqslant \mathcal{O}(n)$ | 282 | 298 | 270 | 250 | 240 |
| $\leqslant \mathcal{O}(n^2)$ | 345 | 376 | 336 | 300 | 270 |
| $\leqslant \mathcal{O}(n^3)$ | 356 | 383 | 345 | 306 | 278 |

Table 1: Comparison of tools on ITS benchmarks.

cessors allow TCT-LCTRS to derive sublinear bounds. This is the case for all problems where PUBS derives a (precise) logarithmic bound, such as the examples `divByTwo` and `direct_n_log_n`. (KoAT and CoFloCo do not support sublinear bounds, and hence output linear bounds for these examples.) Moreover, we can precisely analyse subproblems produced by a divide-and-conquer approach like `divide_and_conquer`, where TCT (as well as KoAT) produces the tight linear bound, while CoFloCo fails and PUBS gives an exponential bound. Detailled results, including a complete table and TCT output, are available on-line.[4]

We moreover tested TCT on the set of logic programs collected by Mesnard and Neumerkl [31],[5] restricted to deterministic programs. A list of solved problems is available on-line as well.

## 8 Conclusion

This paper presented the first complexity framework for LCTRSs. We conclude by relating to earlier work in the area, before indicating leads for future research.

*Related work.* In the last decades there has been significant progress in the area of *fully automated* resource analysis, showing that it can be both practicable and scalable, see e.g. [39,1,24,2,36,40,3,18,26,32]. In the following, we indicate related work that directly influenced our framework, or employed similar methods.

Our framework differs from earlier work by Avanzini and Moser [4] in three important respects: first, constraints over arbitrary background theories are supported, second, complexity is not expressed in terms of the size of the initial term but in terms of measure functions, and third, sublinear bounds can be derived. While innermost rewriting is a rather natural restriction for LCTRSs, *call by need* strategies could be considered in the future for LCTRSs, too.

LCTRSs generalise ITSs, the complexity analysis of which is subject to a comprehensive line of research [8,34]. Our approach gracefully extends the alternating time and size bound technique by Brockschmidt *et al.* [8], as the ITS case

[4] See http://cl-informatik.uibk.ac.at/users/swinkler/lctrs_complexity/
[5] See http://www.complang.tuwien.ac.at/cti/bench/.

is fully covered. In addition, we can obtain sublinear bounds, and support further modularization. Moreover, LCTRSs offer native support for full recursion.

Sublinear bounds are beyond the scope of this earlier work, but can be inferred by some other tools. Albert *et al.* [1] apply refinements to linear ranking functions and support sufficient criteria for divide-and-conquer patterns. This allows the tool PUBS to recognize logarithmic and $\mathcal{O}(n \log(n))$ bounds for some problems. Chatterjee *et al.* [9] use synthesis ranking functions extended by logarithmic and exponential terms, making use of an insightful adaption of Farkas' and Handelman's lemmas. The approach is able to handle examples such as mergesort. In contrast to our work this amounts to a whole-program analysis. Further, extensibility to a constraint formalism like LCTRS is unclear. Wang *et al.* [38] present an ML-like language with type annotations, also using the Master Theorem to handle divide-and-conquer-like recurrences. To estimate lower bounds for logic programs based on divide-and-conquer, Debray *et al.* [13] consider non-deterministic recurrence relations and propose a technique to obtain a closed-form bound for some cases.

*Future work.* We see exciting directions for future work both on a theoretical and an application level. Various additional processors can be conceived for our complexity framework, for instance forms of dependency pairs for non-innermost rewriting [34,25], knowledge propagation and narrowing [34].

Simplification systems as, for instance, employed in compiler toolchains (cf. Ex. 3) or SMT solvers constitute a highly relevant application domain, since these routines operate in performance-critical contexts. In order to tackle such systems, techniques for *derivational* complexity of LCTRSs need to be developed.

On the application level, LCTRSs constitute a natural backend for complexity analysis of constraint logic programs, since constraints can be natively expressed. Our experiments with logic programs did not take backtracking into account, but suitably adapting the transformational frameworks as established by Giesl *et al.* [23] to LCTRSs, this is not a showstopper: There the authors provide an automated complexity and termination analysis of full Prolog programs. In particular, the aforementioned restriction to deterministic programs can be overcome. We thus plan to support CLP as a frontend of our analysis, possibly taking into account *labelling strategies* that control the instantiation of query terms. We furthermore plan to support C programs as a frontend. C programs with integers, as considered in the Termination Competition[6] can be expressed as ITSs. LCTRSs offer more flexibility and can support also strings and floats, as the respective theories are supported by SMT solvers. Just like for the case of CLP, this requires the development of suitable complexity-reflecting transformations. More experiments are planned to evaluate our method on (constrained) logic programs [31] and problems from the software competition.[7]

[6] http://termination-portal.org/
[7] https://sv-comp.sosy-lab.org/

## References


1. E. Albert, P. Arenas, S. Genaim, and G. Puebla. Automatic inference of upper bounds for recurrence relations in cost analysis. In *Proc. 15th SAS*, volume 5079 of *LNCS*, pages 221–237, 2008. https://doi.org/10.1007/978-3-540-69166-2_15.
2. E. Albert, S. Genaim, and A. N. Masud. On the inference of resource usage upper and lower bounds. *ACM TOCL*, 14(3):22, 2013. https://doi.org/10.1145/2499937.2499943.
3. M. Avanzini, U. Dal Lago, and G. Moser. Analysing the complexity of functional programs: Higher-order meets first-order. In *Proc. 20th ICFP*, pages 152–164. ACM, 2015. https://doi.org/10.1145/2784731.2784753.
4. M. Avanzini and G. Moser. A combination framework for complexity. *Inf. Comput.*, 248:22–55, 2016. https://doi.org/10.1016/j.ic.2015.12.007.
5. M. Avanzini, G. Moser, and M. Schaper. TcT: Tyrolean Complexity Tool. In *Proc. 22nd TACAS*, volume 9636 of *LNCS*, pages 407–423. Springer, 2016. https://doi.org/10.1007/978-3-662-49674-9_24.
6. F. Baader and T. Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. https://doi.org/10.1017/CBO9781139172752.
7. R. Bagnara and F. Mesnard. Eventual linear ranking functions. In *Proc. 15th PPDP*, pages 229–238, 2013. https://doi.org/10.1145/2505879.2505884.
8. M. Brockschmidt, F. Emmes, S. Falke, C. Fuhs, and J. Giesl. Analyzing runtime and size complexity of integer programs. *ACM Trans. Program. Lang. Syst.*, 38(4):13:1–13:50, 2016. https://doi.org/10.1145/2866575.
9. K. Chatterjee, H. Fu, and A. K. Goharshady. Non-polynomial worst-case analysis of recursive programs. In *Proc. 29th CAV*, volume 10427 of *LNCS*, pages 41–63, 2017. https://doi.org/10.1007/978-3-319-63390-9_3.
10. Ş. Ciobâcă and D. Lucanu. A coinductive approach to proving reachability properties in logically constrained term rewriting systems. In *Proc. 9th IJCAR*, volume 10900, pages 295–311, 2018. https://doi.org/10.1007/978-3-319-94205-6_20.
11. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, editors. *All About Maude—A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, volume 4350 of *LNCS*. Springer, 2007. https://doi.org/10.1007/978-3-540-71999-1.
12. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *Proc. 14th TACAS*, volume 4963 of *LNCS*, pages 337–340, 2008. https://doi.org/10.1007/978-3-540-78800-3_24.
13. S. K. Debray, P. López-García, M. V. Hermenegildo, and N. Lin. Lower Bound Cost Estimation for Logic Programs. In *Proc. 14th ILPS*, pages 291–305, 1997. https://doi.org/10.7551/mitpress/4283.003.0035.
14. B. Dutertre. Yices 2.2. In *Proc. 26th CAV*, volume 8559 of *LNCS*, pages 737–744, 2014. https://doi.org/10.1007/978-3-319-08867-9_49.
15. S. Falke, D. Kapur, and C. Sinz. Termination analysis of C programs using compiler intermediate languages. In *Proc. 22nd RTA*, volume 10 of *LIPIcs*, pages 41–50, 2011. https://doi.org/10.4230/LIPIcs.RTA.2011.41.
16. A. Flores-Montoya. Upper and lower amortized cost bounds of programs expressed as cost relations. In *Proc. 21st FM*, volume 9995 of *LNCS*, pages 254–273, 2016. https://doi.org/10.1007/978-3-319-48989-6_16.
17. A. Flores-Montoya. *Cost Analysis of Programs Based on the Refinement of Cost Relations*. PhD thesis, Universität Darmstadt, 2017.

18. F. Frohn and J. Giesl. Complexity analysis for Java with AProVE. In *Proc. 13th CAV*, volume 10510 of *LNCS*, pages 85–101, 2017. https://doi.org/10.1007/978-3-319-66845-1_6.
19. C. Fuhs, J. Giesl, M. Plücker, P. Schneider-Kamp, and S. Falke. Proving termination of integer term rewriting. In *Proc. 20th RTA*, volume 5595 of *LNCS*, pages 32–47, 2009. https://doi.org/10.1007/978-3-642-02348-4_3.
20. C. Fuhs, C. Kop, and N. Nishida. Verifying procedural programs via constrained rewriting induction. *ACM TOCL*, 18(2):14:1–14:50, 2017. https://doi.org/10.1145/3060143.
21. Y. Furuichi, N. Nishida, M. Sakai, K. Kusakari, and T. Sakabe. Approach to procedural program verification based on implicit induction of constrained term rewriting systems. *IPSJ Trans. Inf. and Syst.*, 1(2):100–121, 2008. In Japanese.
22. 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. *J. Autom. Reasoning*, 58(1):3–31, 2017. https://doi.org/10.1007/s10817-016-9388-y.
23. 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 *Proc. 14th PPDP*, pages 1–12. ACM Press, 2012. https://doi.org/10.1007/978-3-642-38197-3_1.
24. S. Gulwani. SPEED: Symbolic complexity bound analysis. In *Proc. 21st CAV*, volume 5643 of *LNCS*, pages 51–62, 2009. https://doi.org/10.1007/978-3-642-02658-4_7.
25. N. Hirokawa and G. Moser. Automated complexity analysis based on the dependency pair method. In *Proc. 4th IJCAR*, volume 5195 of *LNCS*, pages 364–379, 2008. https://doi.org/10.1007/978-3-540-71070-7_32.
26. J. Hoffmann, A. Das, and S.-C. Weng. Towards automatic resource bound anaysis for OCaml. In *Proc. 44th POPL*, pages 359–373. ACM, 2017. https://doi.org/10.1145/3009837.
27. C. Kop. Termination of LCTRSs. In *Proc. 13th WST*, pages 59–63, 2013.
28. C. Kop and N. Nishida. Term rewriting with logical constraints. In *Proc. 9th FroCoS*, volume 8152 of *LNAI*, pages 343–358, 2013. https://doi.org/10.1007/978-3-642-40885-4_24.
29. C. Kop and N. Nishida. Constrained term rewriting tool. In *Proc. 20th LPAR*, volume 9450 of *LNAI*, pages 549–557, 2015. https://doi.org/10.1007/978-3-662-48899-7_38.
30. N. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Practical verification of peephole optimizations with Alive. *Commun. ACM*, 61(2):84–91, 2018. https://doi.org/10.1145/3166064.
31. F. Mesnard and U. Neumerkel. Applying static analysis techniques for inferring termination conditions of logic programs. In *Proc. 8th SAS*, volume 2126 of *LNCS*, pages 93–110, 2001. https://doi.org/10.1007/3-540-47764-0_6.
32. G. Moser and M. Schaper. From Jinja bytecode to term rewriting: A complexity reflecting transformation. *Inf. Comput.*, 261(Part):116–143, 2018. https://doi.org/10.1016/j.ic.2018.05.007.
33. N. Nishida and S. Winkler. Loop detection by logically constrained term rewriting. In *Proc. 10th VSTTE*, volume 11294 of *LNCS*, pages 309–321, 2018. https://doi.org/10.1007/978-3-030-03592-1_18.

34. L. Noschinski, F. Emmes, and J. Giesl. Analyzing innermost runtime complexity of term rewriting by dependency pairs. *J. Autom. Reasoning*, 51(1):27–56, 2013. https://doi.org/10.1007/s10817-013-9277-6.
35. A. Podelski and A. Rybalchenko. A complete method for the synthesis of linear ranking functions. In *Proc. 5th VMCAI*, volume 2937 of *LNCS*, pages 239–251, 2004. https://doi.org/10.1007/978-3-540-24622-0_20.
36. A. Serrano, P. López-García, and M. Hermenegildo. Resource usage analysis of logic programs via abstract interpretation using sized types. *TPLP*, 14(4-5):739–754, 2014. https://doi.org/10.1017/S147106841400057X.
37. TeReSe. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.
38. P. Wang, D. Wang, and A. Chlipala. TiML: A functional language for practical complexity analysis with invariants. *Proc. ACM Program. Lang.*, 1(OOPSLA), 2017. https://doi.org/10.1145/3133903.
39. 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. Stenström. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Prog. Lang. Syst.*, 7(3), 2008. https://doi.org/10.1145/1347375.1347389.
40. R. Wilhelm and D. Grund. Computation takes time, but how much? *Commun. ACM*, 57(2):94–103, 2014. https://doi.org/10.1145/2500886.
41. S. Winkler and A. Middeldorp. Completion for logically constrained rewriting. In *Proc. 3rd FSCD*, volume 108 of *LIPIcs*, pages 30:1–30:18, 2018. https://doi.org/10.4230/LIPIcs.FSCD.2018.30.
42. S. Winkler and G. Moser. Runtime complexity analysis of logically constrained rewriting (extended version). Available from http://cl-informatik.uibk.ac.at/users/swinkler/lctrs_complexity/paper.pdf.