Rheinisch-Westfälische Technische Hochschule Aachen
Lehr- und Forschungsgebiet Informatik 2
Programmiersprachen und Verifikation

Automated Complexity Analysis of Term Rewrite Systems

Lars Noschinski

Diplomarbeit
im Studiengang Informatik

vorgelegt der
Fakultät für Mathematik, Informatik und Naturwissenschaften der
Rheinisch-Westfälischen Technischen Hochschule Aachen

im März 2010

Gutachter: Prof. Dr. Jürgen Giesl
Prof. Dr. Peter Rossmanith

Danksagungen

Ich bedanke mich ganz herzlich bei Prof. Giesl für seine Unterstützung und die Möglichkeit, diese Arbeit im Rahmen des AProVE-Projektes durchzuführen und bei Prof. Rossmanith für seine Bereitschaft diese Arbeit zu bewerten.

Mein besonderer Dank gilt Fabian Emmes für die Betreuung und die vielen hilfreichen und anregenden Diskussionen. Auch bei Carsten Fuhs bedanke ich mich für zahlreiche hilfreiche Hinweise. Ich danke auch Carsten Otto, Marc Brockschmidt, Christian von Essen und den übrigen Mitarbeitern und Diplomanden des Lehrstuhls für eine sehr angenehme, (nicht nur Arbeits-)atmosphäre.

Ganz besonders bedanke ich mich bei meiner Freundin und meiner Familie dafür, dass sie immer für mich da sind und mich tatkräftig unterstützen.

Hiermit versichere ich, dass ich die Arbeit selbständig verfasst und keine anderen als die angebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 18. März 2010,

Lars Noschinski

Contents

1. Introduction

By now, software systems are an ubiquitous part of our lives and we depend on them in many ways. Often, we are not aware of those systems, yet even vitally important facilities depend on them: Modern communication systems, banks, traffic control, medical systems, the control of a nuclear reactor. For a private computer mainly used for games and surfing, a failure may only be a mere nuisance, but for other systems, a failure can even have drastic results: All traffic lights on a crossroad switching to green at the same time may result in a crash. A financial system not correctly accounting money can have severe economical consequences. But even for systems not that critical, the costs resulting of faulty software can be very high.

So there is a great incentive for knowing that a system performs as intended. With the ever-growing complexity of such systems, understanding them becomes a difficult task. Testing the systems on known cases is a widely used technique, but fails if cases occur the testers did not thought of. Also, testing can never guarantee correctness, as every sufficiently big system has an effectively infinite number of configurations or inputs.

Another approach are mathematical proofs. For this, one needs a mathematical model in which this proof is established. It is necessary that the model accurately describes all aspects we are interested in. Yet, it must be concise enough, so a proof can be carried out. Exactly modeling real world systems is often not possible, so the model will only be an approximation. If a model has been found, the requirements can be formulated in this model. Care must be taken that the specification really models exactly what a correct system is intended to do.

The last step is proving that the system fulfills the specification. Doing this by hand is error-prone and often not feasible, due to the complexity of the system. Hence, strong automated analysis is a worthwhile goal. Two well studied characteristics are partial correctness and termination. If a program is partially correct, every result computed by this program is correct. A terminating program is one which produces a result for every input. But not only whether a program produces a result is important, but also how long it takes. Hence, the topic of this diploma thesis is to automatically prove upper bounds for the runtime of a program. All techniques described in this paper were implemented in the automated termination (and now also complexity) prover AProVE [GST06].

Real programming languages are often not specified with formal rigor or the sheer complexity of the language makes a direct analysis hardly manageable. Therefore, one chooses a simpler theoretical system in which analysis is performed and transforms the original program to this system, while preserving the relevant properties.

Term Rewriting is a branch of theoretical computer science which combines deductive proving, programming and elements of logic. It may be used for the analysis of systems from equational logic. In this context, the rules of a term rewrite system can be conceived

as directed equations and the syntactic rewrite relation can be used to automatically deduce semantic properties of the corresponding equation system.

On the other hand, Term Rewrite Systems (TRS) are a simple, albeit Turing-complete, programming calculus, which shares a big similarity with functional languages. Programs which make use of pattern matching can often be easily translated. Consider the following Haskell program computing the length of a list:

```
data Nats = Zero | Succ Nats

length [] = Zero
length (x : y) = Succ (length y)
```

This is essentially the same function as the TRS $\mathcal{R}_{\mathsf{length}}$ (both in terms of resulting term and the number of evaluation steps):

$$\begin{aligned} \mathsf{length}(\mathsf{Nil}) &\to \mathsf{Zero} \\ \mathsf{length}(\mathsf{Cons}(x, y)) &\to \mathsf{Succ}(\mathsf{length}(y)) \end{aligned}$$

Our mental model is that the two *rules* of $\mathcal{R}_{\mathsf{length}}$ define a function length, which can be computed by applying the rules to a term and we call length a *defined symbol*. The other symbols in this system are *constructor symbols*. They cannot be evaluated by themselves and hence can be conceived as data.

These properties and their conceptual simplicity make term rewrite systems a well-suited tool for the analysis of programs. Recent efforts successfully employ term rewriting techniques to automatically prove termination for different real-world languages: Haskell [GSST06], Prolog [SGST09] and Java Bytecode [OBEG10].

Once one has established that a TRS is terminating, the logical next step is to study the evaluation complexity: Given a term t, what is the longest sequence of evaluation steps which can be performed starting with t? As usual, we are interested in the connection between the size of the input and the length of the sequence. For *Derivational Complexity*, one considers an arbitrary formed input. The analysis of *Runtime Complexity* restricts the start terms to a specific form, the so-called *Basic Terms*: Only the outermost symbol may be a defined one, all other symbols in the term must be constructors. So the runtime complexity of TRS is the complexity of a single function call, while the derivational complexity is the complexity of an arbitrary expression consisting of the functions defined in the TRS.

There are at least two view points for this analysis. One possibility is to investigate which complexity bounds are imposed by different termination techniques. This leads to characterizations of complexity classes by rewrite techniques [BCMT99b]. On the other hand, one can use those techniques to automatically derive upper bounds for algorithms specified by term rewrite systems. Early work in termination analysis concentrated on *direct termination techniques*, proving the termination of a system in one step: Simplification orders like the *Lexicographic Path Order* [Der87], *Knuth Bendix Orders* [ZHM09; KB70] and orders defined by interpretations like *Polynomial Orders* [CMTU05] [Lan79]. The complexity bounds derived by such techniques are of a rather

theoretical interest: Doubly exponential for polynomial interpretations [HL89], multiply recursive for lexicographic path orderings [Wei95]. Recently, a path order guaranteeing polynomial complexity was presented by Avanzini and Moser [AM08]. Sufficiently restricted polynomial interpretations bound the complexity in their degree [BCMT99a], but even unrestricted polynomial interpretations are often too weak when applied directly: The obviously terminating rule $\mathsf{f}(\mathsf{0},\mathsf{1},x) \to \mathsf{f}(x,x,x)$ cannot be oriented strictly by any polynomial order.

In the termination community, focus has since shifted to transformation techniques like semantic labeling [Zan95] and dependency pairs [AG00], which improve the power of classic termination techniques considerably.

The original Dependency Pair Method does not preserve polynomial complexities. Hence Hirokawa and Moser [HM08a] introduced a variant called Weak Dependency Pairs (see also [HM08b]) and presented adapted versions of the Reduction Pair and Dependency Graph processors. Unfortunately, their techniques require the use of the very restricted class of *Strongly Linear Interpretations* (SLI).

Hence, in this thesis we present *Complexity Dependency Tuples*, a new variant of Dependency Pairs, which preserves an upper bound for runtime complexity of TRSs with an eager evaluation strategy. For confluent systems, the transformed system even has the exactly same runtime complexity. We also developed an extended variant which exactly preserves upper and lower bounds for all systems. Unlike to the Weak DP approach by Hirokowa and Moser, no SLI are needed. Our approach makes more informations about the function call structure explicit; hence our Dependency Graph admits well-known graph transformations like Instantiation, Narrowing and Rewriting [GTSF06]. This techniques are embedded in a general framework, which allows easy extension through additional processors.

This thesis is organized as follows: In the first chapter, we motivate our work and recapitulate some basic notions. The second chapter introduces our main contribution: A framework for proving the runtime complexity of TRSs based on our newly-developed Complexity Dependency Tuples (CDT). Also, two elementary techniques (called *Processors* in our framework) are introduced: The Usable Rules Processor (Section 2.4) removes rules not needed for the further analysis. The Reduction Pair Processor (Section 2.5) can be used iteratively to prove upper bounds for single tuples, hence simplifying the remaining problem. In Chapter 3, we introduce the SCC Split Processor, our equivalent to the original Dependency Graph Processor. The other big contribution of this chapter are the graph transformation techniques, which greatly increase the power of our approach. The chapter on annotated CDTs finally presents an extension of CDTs to achieve an exact transformation of the original system. Finally, we give experimental results, comparing an implementation of our techniques against tools implementing existing techniques. We conclude with an discussion of topics interesting for further investigation.

2. Complexity Dependency Tuples

We start with a short introduction to term rewriting. For an in-depth treatment of the topic, see also [Ohl02] or [BN98].

Definition 2.1 (Term)**.** Let Σ be a finite signature of function symbols and $\mathcal{V}$ be a (possibly infinite) set of *variables*. Each function symbol $f \in \Sigma$ has an unique arity $\mathrm{ar}_f \in \mathbb{N}$. The set of terms over Σ and $\mathcal{V}$ is the minimal set $\mathcal{T}(\Sigma, \mathcal{V})$ such that:

- $\mathcal{V} \subseteq \mathcal{T}(\Sigma, \mathcal{V})$ and
- $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$ if $f \in \Sigma$ with arity n and $t_1, \ldots, t_n \in \mathcal{T}(\Sigma, \mathcal{V})$.

The set of variables occurring in a term t is denoted by $\mathcal{V}(t)$. If $t = f(t_1, \ldots, t_n)$, then f is the *root symbol* of t, denoted by $\mathrm{root}(t)$. The terms $t_1, \ldots, t_n$ are the *arguments* of t. The *size* $|t|$ of a term t is the number of function symbols in t, i.e., $|x| = 0$ for $x \in \mathcal{V}$ and $|f(t_1, \ldots, t_n)| = 1 + \sum_{1 \leq i \leq n} |t_n|$.

A *Term Rewrite System* is a finite set of rules $\mathcal{R}$. A *rule* is a pair of terms $l \to r$ such that the *left-hand side* (or lhs) l is not a variable and all variables of the *right-hand side* (or rhs) r occur also in l, i.e., $\mathcal{V}(r) \subseteq \mathcal{V}(l)$. The latter is called the *variable condition*. A rule not satisfying the variable condition is called a *generalized rule*.

We understand a TRS as a program and the rules as definitions of functions. Hence we split the signature in two parts.

Definition 2.2 (Defined Symbols and Constructor Symbols)**.** We call a function symbol $f \in \Sigma$ *defined* (with regard to a set of rules $\mathcal{R}$) if there is a rule $l \to r \in \mathcal{R}$ such that f is the root symbol of l. The set of defined function symbols is denoted with $\mathcal{D}_\mathcal{R}$.

The remaining symbols are called *constructor symbols* and their set is denoted by $\mathcal{C}_\mathcal{R}$.

Often, we need to decompose terms. For this, we need the notion of a *position*.

Definition 2.3 (Position)**.** A *position* is a word over $\mathbb{N}^*$. The empty word is denoted by ε. For a term t, the subterm $t|_\pi$ at position π is

$$t|_\pi := \begin{cases} t & \text{if } \pi = \varepsilon \\ t_i|_{\pi'} & \text{if } t = f(t_1, \ldots, t_n) \text{ and } \pi = i.\pi' \end{cases}$$

The set of positions of t is denoted by $\mathrm{Pos}(t)$. A position π of a term t is called *defined* (w.r.t. a TRS $\mathcal{R}$) if the root symbol of $t|_\pi$ is defined.

There are two orders on $\mathbb{N}^*$: We write $\pi < \tau$ if $\tau = \pi\kappa$ for a non-empty κ. We write $\leq$ for the reflexive closure of $<$. If neither $\pi \leq \tau$ nor $\tau \leq \pi$, then π and τ are *independent* or *parallel*, denoted by $\pi \perp \tau$. The other order is the *lexicographic order*: $\pi <_{\mathrm{lex}} \tau$ if and only if

- $\pi = \varepsilon \neq \tau$ or
- $\pi = i.\pi'$, $\tau = j.\tau'$ and $i <_{\mathbb{N}} j$ (where $<_{\mathbb{N}}$ is the usual order on the naturals) or
- $\pi = i.\pi'$, $\tau = i.\tau'$ and $\pi' <_{\text{lex}} \tau'$.

For terms s, t and a position $\pi \in \text{Pos}(t)$, $t[s]_\pi$ is the term derived from t by replacing the subterm at position π by s.

Definition 2.4 (Substitution and Matching)**.** A substitution σ is a map $\mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$. In contrast to the usual definition, we allow σ to have an infinite domain. Substitutions are extended to terms in the obvious manner, i.e., $\sigma(f(t_1, \ldots, t_n)) = f(\sigma(t_1), \ldots, \sigma(t_n))$. We usually write $t\sigma$ instead of $\sigma(t)$.

A term t *matches* a term p, if $t = p\sigma$ for some substitution σ. In this case, we say t is an *instance* of p. Two terms s and t *unify* with a substitution σ, if $s\sigma = t\sigma$. The substitution is called the *most general unifier* or *mgu*, if for each substitution μ unifying s and t we have $\mu = \sigma\tau$ for some substitution τ.

Now, s rewrites to t with rule $l \to r$ at position π, denoted by $s \xrightarrow{\pi}_{l \to r} t$ if $s = C[l\sigma]_\pi$ and $t = C[r\sigma]_\pi$ for some substitution π and some context C. In this case, we call $l\sigma$ a *redex* (reducible expression). Now, $s \to t$ is called a *rewrite step*. We say a term t is a *normal form* of s with regard to a TRS $\mathcal{R}$, if $s \to^* t$ and t contains no $\mathcal{R}$-redex. A system is *weakly terminating*, if each term has a normal form and *(strongly) terminating* if there are no infinite reductions. It is *confluent*, if $t_1 \leftarrow^* s \to^* t_2$ implies $t_1 \to^* u \leftarrow^* t_2$. In particular, in a confluent system each term has at most one normal form.

A rewrite step is *innermost*, denoted by $s \xrightarrow{\text{i}} t$, if all arguments of $s\sigma$ are in normal form with regard to $\mathcal{R}$. The definition of *innermost termination* is analogous to the definition above. The transitive closure of a rewrite relation is denoted by a *, e.g. $\to^*$ or $\xrightarrow{\text{i}}{}^*$.

A term starting with a defined function symbol may be evaluated and hence our mental model is that of a function call. On the other hand, a term consisting purely of constructors is always static and hence can be conceived as data. This mental model works best if the TRS is a non-overlapping constructor system. In a *constructor system*, the arguments of all left-hand sides are constructor terms. A system is called *non-overlapping*, if no left-hand sides l_1, l_2 exist such that l_1 unifies with a subterm of l_2. We will see that the techniques presented here work equally well for the weaker notion of a confluent system.

2.1. Complexity of a TRS

An obvious way to define the complexity of a TRS is to measure the length of a rewrite sequence against the size of an arbitrary start term. This is called the *derivational complexity* of a Term Rewrite System. However, this measurement does not measure the complexity of the functions defined by a TRS, but the complexity of all possible combinations of those functions.

Example 2.5. Consider the following simple rewrite system computing the double of a natural number.

$$\mathsf{double}(\mathsf{0}) \to \mathsf{0} \tag{2.1}$$
$$\mathsf{double}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{double}(x))) \tag{2.2}$$

Obviously, doubling a number in unary notation has linear complexity (each s symbol must be replaced by two s symbols) and it is easy to see that the second rule can only be applied n times to a term of the form $\mathsf{s}^n(0)$ (s^n denotes applying s n-times).

But the derivational complexity of the above system is in fact exponential: The start term $\mathsf{double}^n(\mathsf{s}(\mathsf{0}))$ computes 2^n and hence has a exponential derivation length. A similar, but more complex example was given by Cichon and Lescanne [CL92] in an analysis of TRSs for number theoretic functions.

When analyzing algorithms, one normally considers only the complexity of the algorithm applied to data, i.e., applied to something which does not evaluate by itself. To adapt this mental model to term rewrite systems, we will consider only start terms where only the root symbol denotes a function. This notion was introduced by [HM08a].

Definition 2.6 (Derivation length)**.** Let t be a term and $\rightharpoonup$ a rewrite relation. Then the *derivation length* of t with regard to $\rightharpoonup$ is the length of the longest $\rightharpoonup$-sequence starting with t. This is denoted by $\mathrm{dl}(t, \rightharpoonup)$. If the derivation is infinite, we set $\mathrm{dl}(t, \rightharpoonup) := \infty$.

Arithmetic on $\mathbb{N}_0 \cup \{\infty\}$ is defined as usual: $n + \infty = \infty$ for all $n \in \mathbb{N}_0$ and $n \cdot \infty = \infty$ for all $n > 0$. The product $0 \cdot \infty$ is undefined. We have $n < \infty$ for all $n \in \mathbb{N}_0$ and $\infty \leq \infty$.

Definition 2.7 (Runtime Complexity)**.** Let T be a set of terms and $\rightharpoonup$ a rewrite relation. Then

$$\mathrm{rc}(n, T, \rightharpoonup) := \max\{\mathrm{dl}(t, \rightharpoonup) \mid t \in T, |t| \leq n\}$$

is the *runtime complexity* of $\rightharpoonup$ with regard to the set of start terms T.

For $T = \mathcal{T}(\Sigma, \mathcal{V})$, rc is the *derivational complexity function*. We get the complexity measurement we are interested in, if we choose T to be the set of basic terms.

Definition 2.8 (Basic Terms)**.** A term t is called basic, if the only defined symbol in t is the root symbol. Formally, we define

$$\mathcal{T}_{B,\mathcal{R}} := \{f(t_1, \ldots, t_n) \mid f \in \mathcal{D}_\mathcal{R} \text{ and } t_i \in \mathcal{T}(\mathcal{C}_\mathcal{R}, \mathcal{V}) \text{ for all } 1 \leq i \leq n\}$$

as the set of basic terms.

Advanced terms are a slight generalization of basic terms: They allow arbitrary normal forms instead of constructor terms as arguments. Such terms will often occur as intermediate steps.

Definition 2.9 (Advanced Terms)**.** A term t is called advanced, if all arguments are in normal form. Formally, we define

$$\mathcal{T}_{A,\mathcal{R}} := \{f(t_1 \dots, t_n) \mid f \in \mathcal{D}_{\mathcal{R}} \text{ and } t_i \text{ in } \mathcal{R}\text{-normal form for all } 1 \leq i \leq n\}$$

as the set of advanced terms.

Throughout this thesis, if we talk about runtime complexity without specifying the set of start terms, we refer to the runtime complexity on basic terms. In this case, we will often write just $\mathrm{rc}(n, \rightharpoonup)$ instead of $\mathrm{rc}(n, \mathcal{T}_B, \rightharpoonup)$.

As can be seen in the preceding explanations, the complexity of a TRS depends on the choice of the rewrite relation. In many cases, the innermost rewrite relation yields exponentially better complexity bounds. This is demonstrated by the following example describing a tree construction.

Example 2.10.

$$\mathsf{tree}(\mathsf{0}) \to \mathsf{leaf} \tag{2.3}$$
$$\mathsf{tree}(\mathsf{s}(x)) \to \mathsf{makeNode}(\mathsf{tree}(x)) \tag{2.4}$$
$$\mathsf{makeNode}(x) \to \mathsf{node}(x, x) \tag{2.5}$$

Using an innermost strategy, this TRS has a linear runtime complexity, as only fully evaluated terms are duplicated. But if we are going to always evaluate the makeNode rule first, we need exponentially many steps to reach a normal form.

Most programming languages use a strict evaluation strategy (which corresponds to innermost rewriting). Even the transformation of Haskell, which uses a lazy evaluation strategy, to TRSs in AProVE ([GSST06]) uses an innermost strategy. Therefore, we will mostly restrict our work to innermost rewriting.

2.2. Dependency Tuples

Let us introduce some notation first. For a function symbol f in the signature Σ, the fresh function symbol $\mathsf{f}^\sharp$ is the tuple symbol of f. The set of tuple symbols is denoted by $\Sigma^\sharp$. For examples, we reuse the original notation of [AG00] and denote the tuple symbols with uppercase letters instead. For a term $t = \mathsf{f}(t_1, \dots, t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$, we call $t^\sharp = \mathsf{f}^\sharp(t_1, \dots, t_n)$ the *sharped term* of t. The set of all those *sharped terms* is denoted by $\mathcal{T}^\sharp(\Sigma, \mathcal{V})$. For the set of basic sharped terms we write $\mathcal{T}^\sharp_B(\Sigma, \mathcal{V}) := \{t^\sharp \mid t \in \mathcal{T}_B(\Sigma, \mathcal{V})\}$.

The Dependency Pair approach is a powerful method for termination analysis. The idea is to make function calls explicit.

Example 2.11 ([AG00])**.** So for the system $\mathcal{R}$ consisting of the rules

$$\mathsf{minus}(x, \mathsf{0}) \to x \tag{2.6}$$
$$\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{minus}(x, y) \tag{2.7}$$
$$\mathsf{quot}(\mathsf{0}, \mathsf{s}(y)) \to \mathsf{0} \tag{2.8}$$
$$\mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y))) \tag{2.9}$$

the set of dependency pairs $\mathcal{P}$ is defined as

$$\mathsf{MINUS}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{MINUS}(x, y) \tag{2.10}$$
$$\mathsf{QUOT}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{QUOT}(\mathsf{minus}(x, y)) \tag{2.11}$$
$$\mathsf{QUOT}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{MINUS}(x, y). \tag{2.12}$$

Now, to prove termination of $\mathcal{R}$, one can show that it suffices to prove that no infinite chain of dependency pairs exists. Intuitively, a sequence S of pairs $s_1 \to t_1, s_2 \to t_2, \ldots$ is a chain, if there exists a $\mathcal{R} \cup \mathcal{P}$ derivation D such that D starts with an instance of s_1 and S is the sequence of $\mathcal{P}$-steps at the top position in D.

By their modular nature, Dependency Pairs admit a number of transformation techniques like the dependency graph, usable rules, (forward) instantiation, narrowing and more. In this chapter we will explore how to adapt the Dependency Pair approach and related techniques to complexity analysis.

Now, one idea to prove the complexity of a system would be counting the $\mathcal{P}$-steps (instead of just proving there are only finitely many of them). Unfortunately, the standard DP-transformation does not preserve complexity: The maximal length of a chain may be exponentially smaller than the runtime complexity of the system, as can be seen below.

Example 2.12. Consider the following system, which builds a full binary tree of a given depth:

$$\mathsf{tree}(\mathsf{0}) \to \mathsf{leaf} \tag{2.13}$$
$$\mathsf{tree}(\mathsf{s}(x)) \to \mathsf{node}(\mathsf{tree}(x), \mathsf{tree}(x)) \tag{2.14}$$

Obviously, this system has an exponential (innermost) runtime complexity. But if we look at the corresponding set of dependency pairs

$$\mathsf{TREE}(\mathsf{s}(x)) \to \mathsf{TREE}(x), \tag{2.15}$$

it is easy to see that each chain can have at most a length linear in the size of the start term (as the term gets smaller in each iteration).

It is easy to see why this is the case: When we use the pair (2.15), we always consider only one of the possible function calls of the right-hand side of the rule (2.14). Therefore, in each step we discard half the remaining reductions. For a more in-depth investigation of the complexity of the regular dependency pair method see [MSW08].

To prevent the described problem, we need to consider all function calls on the right-hand side of a rule. For this, we extend the definition of Dependency Pairs to *Dependency Tuples*: Instead of single recursive call, now a list of recursive calls may occur on the right-hand side.

Definition 2.13 (Complexity Dependency Tuple)**.** Let $l, r_1, \ldots, r_n \in \mathcal{T}^\sharp(\Sigma, \mathcal{V})$. Then

$$l \to [r_1, \ldots, r_n]$$

is a *Complexity Dependency Tuple* (CDT). Instead of $[r_1, \ldots, r_n]$ we often write $\overline{r}$. If we want to represent the right-hand side of tuple as a term, we tie the elements of the list together by the use of a compound symbol.

Definition 2.14 (Compound symbols)**.** Let Σ be a signature. Let Σ_C be a set of function symbols disjoint to Σ and $\Sigma^\sharp$. Then the elements of Σ_C are called *compound symbols*.

Usually, we are not interested in the exact compound symbol and denote an arbitrary compound symbol by $\textsc{Com}$. So instead of $l \to [r_1, \ldots, r_n]$ we write $l \to \textsc{Com}(r_1, \ldots, r_n)$. Compound symbols are not supposed to occur below other kinds of function symbols. They are only intended to tie together sharped terms (or compounds of sharped terms). We call such terms *well-formed*.

Definition 2.15 (Well-formed terms)**.** A term t is *well-formed*, if it is a compound of sharped terms. They are denoted by $\mathcal{T}_T^\sharp(\Sigma, \mathcal{V})$, which is the minimal set such that

(1) $\mathcal{T}^\sharp \subseteq \mathcal{T}_T^\sharp$ and

(2) $\textsc{Com}(t_1, \ldots, t_n) \in \mathcal{T}_T^\sharp$ if $\textsc{Com}$ is a compound symbol of arity n and $t_1, \ldots, t_n$ are well-formed.

It is easy to see that rewriting a well-formed term yields a well-formed term again. Now we can define the CDT transformation. Instead of creating one dependency pair for each defined subterm of the right-hand side of a rule, we create one dependency tuple which covers all dependency pairs.

Definition 2.16 (CDT Transformation)**.** Let $\mathcal{R}$ be a set of rules over $\mathcal{T}(\Sigma, \mathcal{V})$. Let $l \to r \in \mathcal{R}$ be a rule and $\pi_1 <_{\text{lex}} \cdots <_{\text{lex}} \pi_n$ be the defined positions of r. Then

$$l^\sharp \to [r|_{\pi_1}^\sharp, \ldots, r|_{\pi_n}^\sharp]$$

is the CDT of $l \to r$. The set of dependency tuples of $\mathcal{R}$ is denoted by $\textsc{Dt}_\mathcal{R}$.

Example 2.17. For the system given in 2.12, the set of dependency tuples is given by

$$\mathsf{T}(\mathsf{0}) \to [] \tag{2.16}$$
$$\mathsf{T}(\mathsf{s}(x)) \to [(\mathsf{T}(x), \mathsf{T}(x)]. \tag{2.17}$$

Note that even for rules without a function call on the right-hand side a dependency tuple is created. Later on, we will see that those may be removed without hurting the (asymptotic) complexity of a system.

Nested function calls in a rule are linearized, so we get

$$\mathsf{MINUS}(x, \mathsf{0}) \to [] \tag{2.18}$$
$$\mathsf{MINUS}(\mathsf{s}(x), \mathsf{s}(y)) \to [\mathsf{MINUS}(x, y)] \tag{2.19}$$
$$\mathsf{QUOT}(\mathsf{0}, x) \to [] \tag{2.20}$$
$$\mathsf{QUOT}(\mathsf{s}(x), \mathsf{s}(x)) \to [\mathsf{QUOT}(\mathsf{minus}(x, y)), \mathsf{MINUS}(x, y)] \tag{2.21}$$

for the system in Example 2.11.

Example 2.18. Complexity Dependency tuples do not always exacly model the possible derivations in the original system. Consider the following TRS

$$\mathsf{g}(x) \to x \tag{2.22}$$
$$\mathsf{g}(x) \to \mathsf{a}(\mathsf{f}(x)) \tag{2.23}$$
$$\mathsf{f}(\mathsf{s}(x)) \to \mathsf{f}(\mathsf{g}(x)) \tag{2.24}$$

with the Dependency Tuples

$$\mathsf{G}(x) \to [] \tag{2.25}$$
$$\mathsf{G}(x) \to [\mathsf{F}(x)] \tag{2.26}$$
$$\mathsf{F}(\mathsf{s}(x)) \to [\mathsf{F}(\mathsf{g}(x)), \mathsf{G}(x)]. \tag{2.27}$$

This does not exactly model the possible derivations in the original system, because $\mathsf{g}(x)$ can be evaluated independent of $\mathsf{G}(x, y)$ in the last tuple, for example by using the first rule for g, but the second tuple (which corresponds to the second rule) for G. We will further investigate this in Chapter 4. For now, we just like to point out that this is not a concern for problems which are confluent with an innermost rewrite strategy.

Now that we have extended Dependency Pairs to CDTs, we have to find an equivalent to chains. Let us recall their formal definition. When we talk about chains, we assume all pairs respective tuples in them are renamed variable disjoint.

Definition 2.19 (Chain, [GTSF06] Def. 3)**.** Let $\mathcal{P}$, $\mathcal{R}$ be TRSs. A (possibly infinite) sequence of pairs $s_1 \to t_1, s_2 \to t_2, \dots$ from $\mathcal{P}$ is a $(\mathcal{P}, \mathcal{R})$-chain iff there is a substitution σ with $t_i\sigma \to^*_{\mathcal{R}} s_{i+1}\sigma$ for all i. The chain is an innermost $(\mathcal{P}, \mathcal{R})$-chain iff $t_i\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s_{i+1}\sigma$ and $s_i\sigma$ is in normal form w.r.t. $\mathcal{R}$ for all i.

As the right-hand side of a Dependency Tuple is a list of terms instead of a single term, each pair has not a single successor, but (up to) n successors, where n is the number of terms. Hence, we get no linear sequence, but a (possibly) infinite tree. As a first step, we define a linear chain for dependency tuples. To distinguish these from the chains defined above, we will call them *tuple chains* or short *t-chains*.

Definition 2.20 (t-Chain)**.** Let $\mathrm{D_T}$ be a set of dependency tuples, $\mathcal{R}$ a TRS. A (possibly infinite) sequence of tuples $s_1 \to [t_{1,1}, \dots, t_{1,n_1}], s_2 \to [t_{2,1}, \dots, t_{2,n_2}], \dots$ is a $(\mathrm{D_T}, \mathcal{R})$-*t-chain* iff there is a substitution σ and a (possible infinite) sequence $e_1, e_2, \dots$ of natural numbers, with $t_{i,e_i}\sigma \to^*_{\mathcal{R}} s_{i+1}$ for all i. The chain is an *innermost* $(\mathrm{D_T}, \mathcal{R})$-*t-chain* iff $t_{i,e_i}\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s_{i+1}$ and $s_i\sigma$ is in normal form w.r.t. $\mathcal{R}$ for all i.

We write $s \to \overline{t}\ \langle i \rangle\ u \to \overline{v}$, if the i-th rhs of $s \to \overline{t}$ was used for the chain $s \to \overline{t}, u \to \overline{v}$.

This definition actually is equivalent to the classic definition of chains, but moves the selection of the function calls from the transformation of the system into the definition of the t-chain.

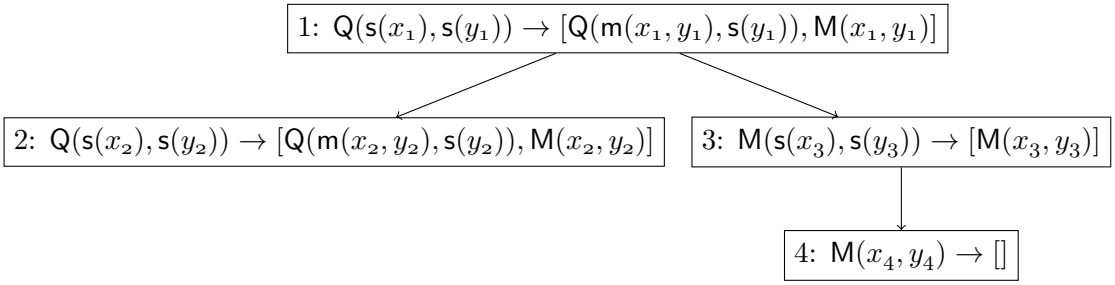
Definition 2.21 (Chain Tree)**.** Let DT be a set of dependency tuples, $\mathcal{R}$ a TRS. A (possibly infinite) tree of tuples from DT is an *(innermost)* $(\mathrm{DT}, \mathcal{R})$*-chain tree*, if there exists a substitution σ, such that each path is an (innermost) $(\mathrm{DT}, \mathcal{R})$-t-chain with this substitution and for each node $s_0 \to \overline{t_0}$ and its successors $s_i \to \overline{t_i}$ with $1 \leq i \leq n_0$ holds: $s_0 \to \overline{t_0}\ \langle i \rangle\ s_i \to \overline{t_i}$.

Such a chain tree *starts with a term* u, if $s \to [t_1, \ldots, t_n]$ is the root node and $u = s\sigma$. A *root chain* is t-chain starting in the root node of the chain tree.

Example 2.22 ($(\mathrm{DT}, \mathcal{R})$-chain tree)**.** Let $(\mathrm{DT}, \mathcal{R})$ be the quot-minus system from Examples 2.11 and 2.17. Then for the substitution

$$\{x_1/\mathsf{s}(\mathsf{s}(0)),\ y_1/\mathsf{s}(0),\ x_2/0,\ y_2/\mathsf{s}(0),\ x_3/\mathsf{s}(0),\ y_3/0,\ x_4/\mathsf{s}(0),\ y_4/0)\}.$$

the following tree is a maximal $(\mathrm{DT}, \mathcal{R})$-chain tree.



Definition 2.23. Let DT be a set of Dependency Tuples, $\mathcal{S}$ a subset of DT, $\mathcal{R}$ a TRS and $t^\sharp \in \mathcal{T}^\sharp(\Sigma, \mathcal{V})$. Then, by $|\mathrm{MCT}(t)|_{\mathcal{S}}$, we denote the maximal number of nodes from $\mathcal{S}$ occurring in an innermost $(\mathrm{DT}, \mathcal{R})$-chain tree starting with t. Again, the size of an infinite tree is denoted by ∞.

If we set $\mathcal{S} = \mathrm{DT}$, then $|\mathrm{MCT}(t)|_{\mathcal{S}}$ is the size of the maximal innermost chain tree starting with t. We will need the set $\mathcal{S}$ as, in contrast to termination analysis, we often will not be able to remove tuples completely. Instead we will remove them from $\mathcal{S}$ only, to mark that they do not need be explicitely counted anymore.

Remark 2.24. The $\mathcal{S}$-size of the biggest chain tree of a term $t^\sharp \in \mathcal{T}^\sharp(\Sigma, \mathcal{V})$ corresponds to the maximal number of $\mathcal{S}$-steps in a $\xrightarrow{\mathsf{i}}_{\mathrm{DT} \cup \mathcal{R}}$-derivation of $t^\sharp$.

Proof. If we understand tuples as rules, then tuple symbols may only occur directly below compound symbols, so each DT-step corresponds to a node in a chain-tree. □

2.3. Framework

The original approach to Dependency Pairs has been formalized in a modular framework. This framework consist of processors which take a dependency pair problem as input and return zero or more new problems as output. The processors are independent in the sense that they do not need to care about how the input problem was generated or

transformed. So integrating new techniques is easily possible; one only needs to check the correctness of the new processor, not of the whole framework extended with the new processor.

In this section we present a similar approach for Complexity Dependency Tuples. For our analysis, we are mostly interested in the asymptotic complexity of a problem, i.e., we are not interested in a concrete function f describing the complexity of a system, but rather in the behavior of this function as the size of the start term approaches infinity. Often the Landau notation is used for this: For example, one writes $f(n) \in \mathcal{O}(g(n))$ if $f(n)$ does not grow substantially faster than $g(n)$ as n approaches infinity. This notation allows to ignore if the growth of two functions differs only by a constant factor. In particular, if f and g are polynomial functions, we have $f(n) \in \mathcal{O}(g(n))$ if and only if the degree of f is at most the degree of g. We choose a related, but slightly different notation for two reasons: First, we want to be able to represent informations as *finite runtime complexity* (i.e., the system is terminating) or *polynomial complexity with unknown degree*, which are not readily representable in Landau notation. Also, we want to perform arithmetic on those values.

Definition 2.25 (Asymptotic Domain)**.** Let

$$\mathfrak{D} := \{\textsc{Poly}(0), \textsc{Poly}(1), \textsc{Poly}(2), \ldots, \textsc{Poly}(?), \textsc{Finite}, \textsc{Infinite}\}$$

be the *domain of asymptotic complexities* with order

$$\textsc{Poly}(0) < \textsc{Poly}(1) < \textsc{Poly}(2) < \ldots < \textsc{Poly}(?) < \textsc{Finite} < \textsc{Infinite}$$

and $\leq$ the reflexive closure of $<$.

We define the conversion function ι such that for a function $f : \mathbb{N} \to \mathbb{N}$ we have

$$\iota(f) := \begin{cases} \textsc{Poly}(k) & \text{if } f(n) \text{ is a polynomial of degree } k \\ \textsc{Infinite} & \text{if } f(n) = \infty \text{ for some } n \in \mathbb{N} \\ \textsc{Finite} & \text{else} \end{cases}$$

We will use this asymptotic domain to represent the complexity of a CDT problem in our framework. A value of $\textsc{Poly}(k)$ represents a complexity bounded by a polynomial of degree k, a value of $\textsc{Poly}(?)$ a complexity bounded by an arbitrary polynomial. A value of $\textsc{Finite}$ denotes a terminating system and $\textsc{Infinite}$ is used, if a system is nonterminating. This is related to Landau notation in the following sense:

$$\begin{aligned} \iota(f) = \textsc{Poly}(k) &\quad\Rightarrow f(n) \in \mathcal{O}(n^k) \\ \iota(f) \leq \textsc{Poly}(?) &\quad\Rightarrow f(n) \in \mathcal{O}(n^k) \text{ for some } k \in \mathbb{N} \\ \iota(f) \leq \textsc{Finite} &\quad\Rightarrow f(n) \in \mathcal{O}(n^k) \text{ for some function } g \\ \iota(f) = \textsc{Infinite} &\quad\Rightarrow \text{none of the above} \end{aligned}$$

We define the arithmetic operations on $\mathfrak{D}$ in a way which is compatible with this relation:

Definition 2.26 (Arithmetic on $\mathfrak{D}$)**.** For $x, y \in \mathfrak{D}$, the binary operation $+$ is defined as $x + y = \max(x, y)$ and the binary operation $\cdot$ as $x \cdot y = \textsc{Poly}(k + l)$, if $x = \textsc{Poly}(k)$ and $y = \textsc{Poly}(l)$, and as $x \cdot y = \max(x, y)$ for all other cases. The binary operation $\ominus$ is defined by

$$x \ominus y = \begin{cases} \textsc{Poly}(0) & \text{if } x \leq y \\ x & \text{else.} \end{cases}$$

The definition of $\ominus$ is guided by the following intuition: Let f, g be two functions such that f grows substantially faster than g, i.e., $g(n) \in \mathcal{O}(f(n))$, but $f(n) \notin \mathcal{O}(g(n))$. Then $f - g$ grows essentially as fast as f, i.e., $f(n) \in \mathcal{O}(f(n) - g(n))$. If on the other hand f does not grow substantially faster than g, then $f - g$ may not grow at all, so $f - g \in \mathcal{O}(1)$ is possible (but not necessarily the case). So we have $\iota(f) \ominus \iota(g) \leq \iota(f - g)$ for all functions.

Remark 2.27. The operations $+, \cdot$ on $\mathfrak{D}$ are associative, commutative and distributive.

Now we introduce the notion of a CDT problem.

Definition 2.28 (CDT problem)**.** Let $\mathcal{R}$ be a TRS, $\textsc{Dt}$ be a set of dependency tuples, $\mathcal{S} \subseteq \textsc{Dt}$. Then $(\textsc{Dt}, \mathcal{S}, \mathcal{R})$ is called a *CDT problem*. The *runtime complexity* of a CDT problem is defined as

$$\mathrm{rc}(n, (\textsc{Dt}, \mathcal{S}, \mathcal{R})) := \max\{|\mathrm{MCT}(t^\sharp)|_{\mathcal{S}} \mid t \in \mathcal{T}_B(\Sigma, \mathcal{V}), |t| \leq n\}.$$

The *asymptotic runtime complexity* of a CDT problem is $\mathrm{rc}_{\mathfrak{D}}(P) := \iota(r_P)$ where $r_P(n) := \mathrm{rc}(n, P)$. For a TRS $\mathcal{R}$, the CDT problem $(\textsc{Dt}(\mathcal{R}), \textsc{Dt}(\mathcal{R}), \mathcal{R})$ is the canonical CDT problem.

For this definition to be useful, the runtime complexity of $(\textsc{Dt}(\mathcal{R}), \textsc{Dt}(\mathcal{R}), \mathcal{R})$ should be an upper bound of the runtime complexity of $\xrightarrow{\mathrm{i}}_{\mathcal{R}}$. This means we have to prove that the derivation length of a basic term t w.r.t. to $\mathcal{R}$ is bounded by the size of a maximal chain tree of $t^\sharp$ w.r.t. to $\textsc{Dt}(\mathcal{R})$ and $\mathcal{R}$. For innermost derivations, all arguments are evaluated to normal forms before the root position is evaluated. We can use this to reduce the analysis of an arbitrary term to the analysis of some advanced terms. Let $\mathrm{dl}_u(t, \xrightarrow{\mathrm{i}}_{\mathcal{R}})$ denote the length of the longest $\xrightarrow{\mathrm{i}}_{\mathcal{R}}$ sequence rewriting a term t to the normal form u. Furthermore for $t = f(t_1, \ldots, t_k)$ let

$$U := \{(u_1, \ldots, u_k) \mid u_i \text{ normal form of } t_i\}$$

the set containing all combinations of normal forms of $t_1, \ldots t_n$. Then we have

$$\mathrm{dl}(t, \xrightarrow{\mathrm{i}}_{\mathcal{R}}) = \max_{(u_1,\ldots,u_k) \in U} \left(\mathrm{dl}(f(u_1, \ldots, u_k), \xrightarrow{\mathrm{i}}_{\mathcal{R}}) + \sum_{1 \leq i \leq k} \mathrm{dl}_{u_i}(t|_i, \xrightarrow{\mathrm{i}}_{\mathcal{R}}) \right).$$

This can be approximated by

$$\mathrm{dl}(t, \xrightarrow{\mathrm{i}}_{\mathcal{R}}) \leq \max_{u_1,\ldots,u_k} \mathrm{dl}\, f(u_1, \ldots, u_k) + \sum_{1 \leq i \leq k} \mathrm{dl}(t|_i, \xrightarrow{\mathrm{i}}_{\mathcal{R}}).$$

For confluent systems this is even an equality, as normal forms are unique. We now introduce a shorter notation to express the maximum in the above formula.

Definition 2.29. Let $t = f(t_1, \ldots, t_k)$ be a term $\mathcal{R}$ a TRS. For each $1 \leq i \leq k$ let s_i be an $\mathcal{R}$-normal form of t_i such that $\mathrm{dl}(f(s_1, \ldots, s_k), \xrightarrow{\mathsf{i}}_{\mathcal{R}})$ is maximal. Then we write $t{\Downarrow} = f(s_1, \ldots, s_k)$ and say t is an *argument normal form.*

It is easy to see that $t{\Downarrow}$ is an advanced term if t has a defined root symbol.

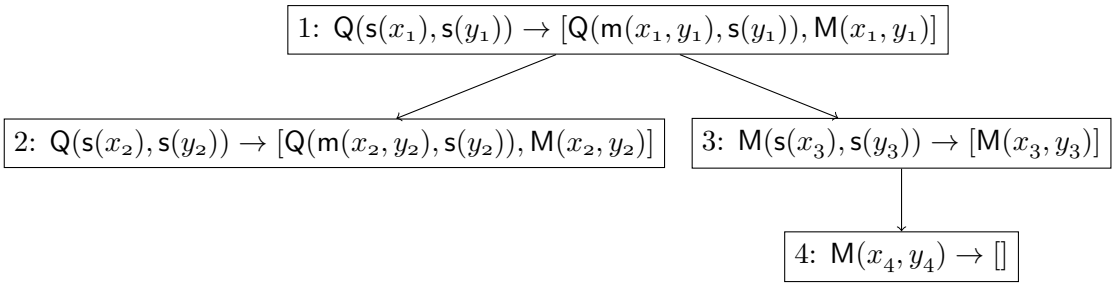
Example 2.30. Consider the TRS given in Example 2.18 and the term $t = \mathsf{f}(\mathsf{g}(\mathsf{s}(x)))$. Now there are two different possibilities to reduce the argument of t to a normal form: If we reduce using the first rule, we get $t_1 = \mathsf{f}(\mathsf{s}(x)$. Using the second rule, we get $t_2 = \mathsf{f}(\mathsf{a}(\mathsf{f}(\mathsf{s}(x)))$. But of these two, only t_1 is in argument normal form, as its derivation length is greather than the derivation length of t_2. Note that an argument normal form does not need to be unique in general.

Now we can show that our definition of the complexity of CDT problem is indeed sensible, as the complexity of a canonical CDT problem is an upper bound for the complexity of the original TRS. We prove this separately for terminating and nonterminating Term Rewrite Systems.

Lemma 2.31. *Let $\mathcal{R}$ be a TRS and $\mathrm{DT}, \mathcal{S} := \mathrm{DT}(\mathcal{R})$. Let $\preccurlyeq$ be $=$ if $\mathcal{R}$ is confluent and $<$ else. If $t \in \mathcal{T}_{A,\mathcal{R}}$ and $\mathcal{R}$ is terminating on t, then we have*

$$\mathrm{dl}(t, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) \preccurlyeq |MCT(t^\sharp)|_{\mathcal{S}}.$$

Example 2.32. Before giving the proof, we will illustrate how this lemma works. We recall the chain tree given in Example 2.22.



The substitution for x_1, y_2 was $\{x_1/\mathsf{s}(\mathsf{s}(0)), y_1/\mathsf{s}(0)\}$, so this chain tree starts with the basic term $\mathsf{Q}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0))), \mathsf{s}(\mathsf{s}(0)))$. Now consider the associated term without tuple symbols. The first reduction step is

$$\mathsf{q}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0))), \mathsf{s}(\mathsf{s}(0))) \xrightarrow{\mathsf{i}} \mathsf{s}(\mathsf{q}(\mathsf{m}(\mathsf{s}(\mathsf{s}(0)), \mathsf{s}(0)), \mathsf{s}(\mathsf{s}(0))))$$

which corresponds to node (1) of the chain tree. In this term, there are two function calls. Both can be counted separately. Before we can evaluate the q symbol, its arguments have to be evaluated in normal form:

$$m(s(s(0)), s(0)) \xrightarrow{\mathsf{i}} \mathsf{m}(\mathsf{s}(0), 0) \xrightarrow{\mathsf{i}} 0$$

These reduction steps corresponds to the nodes (3) and (4) in the graph. Till now, we have 3 reduction steps. Now, when we go from node (1) to node (2) in the graph, the evaluation of $m(x_1, y_1)$ is hidden in the edge:

$$\mathsf{q}(\mathsf{m}(\mathsf{s}(\mathsf{s}(0)),\mathsf{s}(0)),\mathsf{s}(\mathsf{s}(0))) \xrightarrow{\mathsf{i}}{}^{*} \mathsf{q}(\mathsf{s}(0),\mathsf{s}(\mathsf{s}(0)))$$

This is correct, because the evaluation of m (in the $\xrightarrow{\mathsf{i}}{}^{*}$-steps) has already been counted. Now we just have to count the last q-step, which corresponds to node (2):

$$\mathsf{q}(\mathsf{s}(0),\mathsf{s}(\mathsf{s}(0))) \to \mathsf{q}(\mathsf{m}(0,\mathsf{s}(0)),\mathsf{s}(\mathsf{s}(0)))$$

So we have four derivation steps and as many nodes in the corresponding chain tree.

Proof. By induction on $\mathrm{dl}(t, \xrightarrow{\mathsf{i}}_{\mathcal{R}})$. Let $l \to r \in \mathcal{R}$ be a rule. Note that t matches l if and only if $t^\sharp$ matches $l^\sharp$. So, if $\mathrm{dl}(t, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) = 0$, then also $t^\sharp$ in $\textsc{Dt}$-normal form and there is no non-empty $(\textsc{Dt}, \mathcal{R})$-chain tree starting with $t^\sharp$.

Otherwise, there exist a rule $l \to r \in \mathcal{R}$ and a substitution σ such that $t = l\sigma \xrightarrow{\mathsf{i},\varepsilon} r\sigma = u$ and $\mathrm{dl}(t, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) = 1 + \mathrm{dl}(u, \xrightarrow{\mathsf{i}}_{\mathcal{R}})$. By Lemma 2.33 below we have

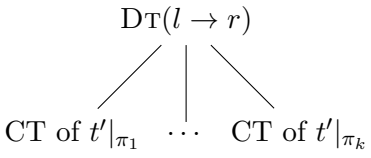
$$\mathrm{dl}(u, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) \lessdot \sum_{\pi \in \mathrm{DPos}(u)} \mathrm{dl}(u|_\pi\Downarrow, \xrightarrow{\mathsf{i}}_{\mathcal{R}})$$

and $\mathrm{dl}(u|_\pi\Downarrow, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) < \mathrm{dl}(t, \xrightarrow{\mathsf{i}}_{\mathcal{R}})$ for all π. As all arguments of $l\sigma$ are in normal form, we have $\mathrm{dl}(u|_\pi, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) = 0$ for all $\pi \in \mathrm{DPos}(u) \setminus \mathrm{DPos}(r)$. Now, by applying the induction hypothesis the (in)equality

$$\mathrm{dl}(t, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) = 1 + \mathrm{dl}(u, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) \lessdot \sum_{\pi \in \mathrm{DPos}(r)} 1 + |\mathrm{MCT}(u|_\pi\Downarrow^\sharp)|_{\mathcal{S}}$$

holds.

Now we will show that there is a chain tree for $t^\sharp$, having $\textsc{Dt}(l \to r)$ as root node and arbitrary non-empty chain trees for the $u|_\pi\Downarrow^\sharp$ with $\pi \in \{\pi_1, \ldots, \pi_k\} = \mathrm{DPos}(r)$ as successors:



We have $\textsc{Dt}(l \to r) = l^\sharp \to [r|^\sharp_{\pi_1}, \ldots, r|^\sharp_{\pi_n}]$ with a substitution σ such that $l\sigma = t$. Hence, $r|_{\pi_i}\sigma = u|_{\pi_i}$ and $r|^\sharp_{\pi_i}\sigma \xrightarrow{\mathsf{i}}{}^{*}_{\mathcal{R}} u|_{\pi_i}\Downarrow^\sharp$. Therefore, if $T_1, \ldots, T_k$ are chain trees for $u|_{\pi_1}\Downarrow^\sharp, \ldots, u|_{\pi_k}\Downarrow^\sharp$, then the tree with root node $\textsc{Dt}(l \to r)$ with successors T_i is a chain tree of t. Hence

$$\mathrm{dl}(t, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) \lessdot |\mathrm{MCT}(t^\sharp)|_{\mathcal{S}} = 1 + \sum_{\pi \in \mathrm{DPos}(r)} |\mathrm{MCT}(u|_\pi\Downarrow^\sharp)|_{\mathcal{S}}.$$

follows.

□

In the lemma above we used the proposition that the derivation length of a term is bounded by the sum of the derivation lengths of the argument normal forms of its subterms. That this is valid is show by the following lemma.

Lemma 2.33. *Let t be a term and $\mathcal{R}$ a TRS. Let $\lessdot$ be $=$ if $\mathcal{R}$ is confluent and $<$ else. Then $\mathrm{dl}(t, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) \lessdot \sum_{\pi \in DPos(t)} \mathrm{dl}(t|_\pi\Downarrow, \xrightarrow{\mathsf{i}}_{\mathcal{R}})$ holds.*

Proof. By induction over $|t|$. For $|t| = 1$, this is obvious as $t\Downarrow = t$. We even have equality. Now consider the case $|t| > 1$ and let the root symbol of t be k-ary. Then, because of the innermost strategy, we have

$$\mathrm{dl}(t, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) \lessdot \mathrm{dl}(t\Downarrow, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) + \sum_{1 \le i \le k} \mathrm{dl}(t|_i, \xrightarrow{\mathsf{i}}_{\mathcal{R}})$$

The $t|_i$ are smaller then t and hence the induction hypothesis can be applied:

$$\begin{aligned}\mathrm{dl}(t, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) &\lessdot \mathrm{dl}(t\Downarrow, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) + \sum_{1 \le i \le k} \sum_{\pi \in \mathrm{DPos}(t|_i)} \mathrm{dl}(t|_{i.\pi}\Downarrow, \xrightarrow{\mathsf{i}}_{\mathcal{R}}) \\ &= \sum_{\pi \in \mathrm{DPos}(t)} \mathrm{dl}(t|_\pi\Downarrow, \xrightarrow{\mathsf{i}}_{\mathcal{R}})\end{aligned}$$

Hence the proposition is shown. □

Lemma 2.34. *Let $\mathcal{R}$ be a set of tuples with $\mathrm{DT}, \mathcal{S} := \mathrm{DT}(\mathcal{R})$. If $\mathcal{R}$ is not innermost terminating on $\mathcal{T}_B$, then there exists a term $t^\sharp$ such that $|MCT(t^\sharp)|_{\mathcal{S}} = \infty$.*

Proof. Let $t_0 \in \mathcal{T}_B$ be a minimal term admitting an infinite $\xrightarrow{\mathsf{i}}_{\mathcal{R}}$ derivation (i.e., all proper subterms of t_0 are $\xrightarrow{\mathsf{i}}_{\mathcal{R}}$-terminating). Then we have $t_0 \xrightarrow{\mathsf{i},>\varepsilon}{}^*_{\mathcal{R}} s_1 = l_1\sigma \xrightarrow{\mathsf{i},\varepsilon} r_1\sigma$ for some rule $l \to r \in \mathcal{R}$. Again $r_1\sigma$ is non-terminating and therefore has a minimal, non-terminating subterm t_1. As the root symbol of t_1 is defined, $l_1\sigma \to t_1$ corresponds to a right-hand side of the dependency tuple of $l_1 \to r_1$. Continuing this way, we get an infinite sequence of dependency tuples.

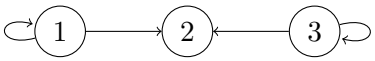
Now, as $s_1 \xrightarrow{\mathsf{i}} r_1\sigma$ is an innermost step, all arguments of s_1 are in normal form and hence are the arguments of $s_1^\sharp$. So, if ν_i is the dependency tuple of $l_1 \to r_1$, then $\nu_1, \nu_2, \ldots$ is an infinite $(\mathrm{DT}, \mathcal{R})$-t-chain. □

Corollary 2.35. *Let $(\mathrm{DT}(\mathcal{R}), \mathrm{DT}(\mathcal{R}), \mathcal{R})$ be a CDT problem. If $\mathcal{R}$ is terminating on all $t \in \mathcal{T}_B$, we have $\iota(r) \le \mathrm{rc}_{\mathfrak{D}}((\mathrm{DT}, \mathcal{S}, \mathcal{R}))$ where $r(n) := \mathrm{rc}(n, \xrightarrow{\mathsf{i}}_{\mathcal{R}})$. If $\mathcal{R}$ is confluent, we even have $\iota(r) = \mathrm{rc}_{\mathfrak{D}}((\mathrm{DT}, \mathcal{S}, \mathcal{R}))$.*

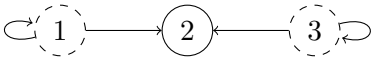
CDT problems can be transformed with *Processors*, which either directly compute the complexity of the problem or generate a new set of problems, from which the complexity of the original system can be computed.

Before we can introduce this concept formally, we need to elaborate on the semantics of the set $\mathcal{S}$ in a $(\mathrm{DT}, \mathcal{S}, \mathcal{R})$ problem. By the definition of the complexity of an CDT problem, we do not need to consider the complexity of the nodes in $\mathrm{DT} \setminus \mathcal{S}$ to compute the complexity of the problem. For our framework, we now have to decide whether we should just ignore the complexity of those nodes or if we assume, that we already know a complexity bound for them. The latter has the advantage that we can use this information to simplify complexity proofs for a system.

Example 2.36. The dependency graph of a CDT problem has the dependency tuples as nodes. Two nodes are connected if there exists a t-chain connecting these two tuples. We will introduce this graph formally in the next chapter. Consider a CDT problem with a dependency graph of the following shape:



Now assume we can iteratively prove a complexity of $c_1 \in \mathfrak{D}$ for node (1) and $c_2 \in \mathfrak{D}$ for node (3). This can be achieved for example by the reduction pair processor presented in Section 2.5. If we remove these two nodes from $\mathcal{S}$, we get the following dependency graph (the nodes not in $\mathcal{S}$ are drawn dashed).



By looking at the graph we see that node (2) cannot occur significantly more often than nodes (1) and (3) together, as only these nodes have edges leading to node (2). This holds true for all CDT problems for which the dependency graph has this shape, regardless of the exact dependency tuples. So the dependency of the whole problem is given by $c_1 + c_2$

To employ this information in our framework, we assume that complexity of the nodes *not* in $\mathcal{S}$ is already known and will be added to the complexity of the nodes in $\mathcal{S}$. Hence we only have to compute a real complexity value if the complexity of the nodes in $\mathcal{S}$ is greater then the complexity of the nodes in $\mathrm{DT} \setminus \mathcal{S}$. Otherwise, we may return an arbitrary value, for example $\textsc{Poly}(0)$, without producing an incorrect result. For this, we introduce the notion of the *complementary complexity*. We call nodes *unknown*, if they are in $\mathcal{S}$ and *known* otherwise.

Definition 2.37. The *complementary complexity* of a CDT problem $(\mathrm{DT}, \mathcal{S}, \mathcal{R})$ is defined as the complexity of the *complementary problem* $(\mathrm{DT}, \mathrm{DT} \setminus \mathcal{S}, \mathcal{R})$:

$$\mathrm{rc}^{\mathsf{c}}(n, (\mathrm{DT}, \mathcal{S}, \mathcal{R})) := \max\{|\mathrm{MCT}(t^\sharp)|_{\mathrm{DT}\setminus\mathcal{S}} \mid t \in \mathcal{T}_B(\Sigma, \mathcal{V}), |t| \leq n\}.$$

The definition for the complementary asymptotic complexity $\mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P)$ is analogous.

Of course, to ensure this leads to a correct result, a processor producing a new problem P must expect to get a complexity value of $\textsc{Poly}(0)$ instead of the real complexity of P, if $\mathrm{rc}_{\mathfrak{D}}(P) \leq \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P)$ holds..

Definition 2.38 (Processors)**.** A *CDT processor* Proc is a function

$$\textsc{Proc}(P) = (f_{\mathsf{up}}, \{P_1, \ldots, P_k\})$$

mapping a CDT problem $P = (\textsc{Dt}, \mathcal{S}, \mathcal{R})$ to a complexity $f_{\mathsf{up}} \in \mathfrak{D}$ and a set of CDT problems $P_1, \ldots, P_k$. A processor is *correct* if

$$\mathrm{rc}_{\mathfrak{D}}(P) \leq f_{\mathsf{up}} + \sum_{1 \leq i \leq k} (\mathrm{rc}_{\mathfrak{D}}(P_i) \ominus \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_i)) + \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P)$$

holds.

Let us elaborate on the correctness inequality above a bit more. Proc computes the result described by

$$f_{\mathsf{up}} + \sum_{1 \leq i \leq k} (\mathrm{rc}_{\mathfrak{D}}(P_i) \ominus \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_i)).$$

The difference $\mathrm{rc}_{\mathfrak{D}}(P_i) \ominus \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_i)$ encodes the expectation, that a processor solving the new problem P_i may return a complexity smaller then $\mathrm{rc}_{\mathfrak{D}}(P_i)$ if $\mathrm{rc}_{\mathfrak{D}}(P_i)$ is bounded by $\mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_i)$. Adding $\mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P)$ is the counterpart to this expectation: It allows the processor to compute a result less or equal to $\mathrm{rc}_{\mathfrak{D}}(P)$, if $\mathrm{rc}_{\mathfrak{D}}(P)$ is bounded by $\mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P)$. How these constraints work is illustrated below.

Example 2.39. Consider the CDT problem $(\textsc{Dt}, \textsc{Dt}, \varnothing)$ with the following set of dependency tuples Dt:

$$\mathsf{F}(\mathsf{s}(x)) \to \mathsf{F}(x) \tag{2.28}$$

$$\mathsf{G}(\mathsf{s}(x)) \to \mathsf{G}(x) \tag{2.29}$$

The runtime complexity of this problem is obviously linear and its complementary complexity is a constant one, that is $\mathrm{rc}_{\mathfrak{D}}(P) = \textsc{Poly}(1)$ and $\mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P) = \textsc{Poly}(0)$. The SCCs of the dependency graph are $\{(2.28)\}$ and $\{(2.29)\}$ and both have linear complexity again, i.e.,

$$\mathrm{rc}_{\mathfrak{D}}(P_1) = \textsc{Poly}(1) = \mathrm{rc}_{\mathfrak{D}}(P_2)$$

where $P_1 = (\textsc{Dt}, \{(2.28)\}, \varnothing)$ and $P_2 = (\textsc{Dt}, \{(2.29)\}, \varnothing)$. However, a processor Proc which splits P in these two new problems, i.e., $\textsc{Proc}(P) = (\textsc{Poly}(0), \{P_1, P_2\})$, would not be correct.

The complementary problem of P_1 is P_2 (and vice versa). Hence the complementary complexity for both P_1 and P_2 is again linear. If we insert those values in the correctness inequality for processors, we get

$$\begin{aligned}\textsc{Poly}(1) &\leq \underbrace{\textsc{Poly}(0)}_{f_{\mathsf{up}}} + \underbrace{(\textsc{Poly}(1) \ominus \textsc{Poly}(1))}_{\mathrm{rc}_{\mathfrak{D}}(P_1) \ominus \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_1)} + \underbrace{(\textsc{Poly}(1) \ominus \textsc{Poly}(1))}_{\mathrm{rc}_{\mathfrak{D}}(P_2) \ominus \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_2)} + \underbrace{\textsc{Poly}(0)}_{\mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P)} \\ &= \textsc{Poly}(0)\end{aligned}$$

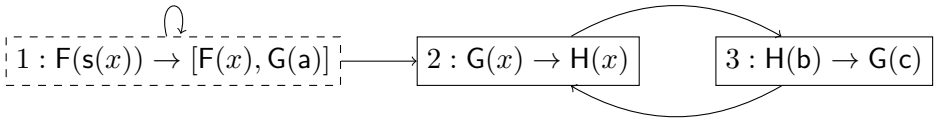
which is obviously not true.

The reason for this is that the dependency on the known nodes is cyclic. Node (2.28) depends on (2.29) for the complexity of its known nodes and vice-versa. See the SccSplitP processor (Definition 3.17) to see how to handle this problem.

Example 2.40. The following example illustrates the advantages of our correctness condition compared to the simpler (but obviously correct) condition

$$\mathrm{rc}_{\mathfrak{D}}(P) \leq f_{\mathsf{up}} + \sum_{1 \leq i \leq k} \mathrm{rc}_{\mathfrak{D}}(P_i)$$

Consider the CDT problem P with the following dependency graph:



Now, the runtime complexity of this problem is linear in the length of the input (as each iteration of tuple 1 calls the second tuple exactly once and one cannot go through the second SCC more then once). On the other hand, we know that in a t-chain both 2 and 3 occur almost the same number of times. So, if we find out that 3 only occurs once in each chain tree, we are tempted to return a constant runtime complexity (and no new problems) for the whole problem.

With the simpler condition, this would not be correct as the constraint $\mathrm{rc}_{\mathfrak{D}}(P) < \textsc{Poly}(0)$ is not fulfilled. However, using the condition of Definition 2.38 we have to fulfill the constraint

$$\mathrm{rc}_{\mathfrak{D}}(P) < \textsc{Poly}(0) + \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P)$$

which is fulfilled, as both the complexity of the first SCC (which equals $\mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P)$) and the complexity of P are linear.

After motivating our definition of a processor, we have to formalize how to combine the results of the processors to a single, upper bound for the complexity of a CDT problem. For this, we build a *proof tree*.

Definition 2.41 (Proof Tree)**.** A *proof tree* is a finite, non-empty tree with nodes (P, f_{up}), where P is a CDT problem such that for each node (P, f_{up}) and its children $(P_1, f_{\mathsf{up}_1}), \ldots, (P_k, f_{\mathsf{up}_k})$ holds: There exists a correct processor $\textsc{Proc}$ such that

$$\textsc{Proc}(P) = (f_{\mathsf{up}}, \{P_1, \ldots, P_k\})$$

holds. A node of the proof tree is also called *proof node*.

The complexity is recursively propagated from the leaves to the root.

Definition 2.42 (Result of a proof node)**.** Let T be a proof tree and N a node in T. Then the *result* of a node (P, f_{up}) is

$$\mathrm{res}(N) := f_{\mathsf{up}} + \mathrm{res}(N_1) + \ldots + \mathrm{res}(N_k)$$

where $N_1, \ldots, N_k$ are the children of N.

Now we need to show that the result of a proof tree is indeed an upper bound for the problem in the root node.

Theorem 2.43 (Correctness of the Proof Tree)**.** *Let T be a finite proof tree with root node (P, f) and $P = (\mathrm{DT}, \mathcal{S}, \mathcal{R})$. Then $\mathrm{rc}_{\mathfrak{D}}(P) \leq \mathrm{res}(P) + \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P)$. In particular, we have $\mathrm{rc}_{\mathfrak{D}}(P) \leq \mathrm{res}(P)$ for $\mathrm{DT} = \mathcal{S}$.*

Proof. We prove the theorem by induction on the depth of T. If the depth is 1, then (P, f_{up}) has no children and

$$\mathrm{rc}_{\mathfrak{D}}(P) \leq f_{\mathsf{up}}(n) + \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(n)P = \mathrm{res}(P) + \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(n)P$$

holds. Else, (P, f_{up}) has children $(P_1, f_{\mathsf{up}_1}), \ldots, (P_k, f_{\mathsf{up}_k})$ and we have

$$\mathrm{rc}_{\mathfrak{D}}(P) \leq f_{\mathsf{up}} + \sum_{1 \leq i \leq k} (\mathrm{rc}_{\mathfrak{D}}(P_i) \ominus \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_i)) + \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P).$$

By the induction hypothesis, $\mathrm{rc}_{\mathfrak{D}}(P_i) \leq \mathrm{res}(P_i) + \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_i)$ holds too, and this implies

$$\begin{aligned}
\mathrm{rc}_{\mathfrak{D}}(P) &\leq f_{\mathsf{up}} + \sum_{1 \leq i \leq k} (\mathrm{rc}_{\mathfrak{D}}(P_i) \ominus \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_i)) + \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P) \\
&\leq f_{\mathsf{up}} + \sum_{1 \leq i \leq k} (\mathrm{res}(P_i) \ominus \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_i)) + \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P) \\
&\overset{(\dagger)}{\leq} f_{\mathsf{up}} + \sum_{1 \leq i \leq k} \mathrm{res}(P_i) + \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P) \\
&= \mathrm{res}(P) + \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P).
\end{aligned}$$

(†): Note that $(x + y) \ominus z = (x \ominus z) + (y \ominus z)$ holds true for all values of $x, y \in \mathfrak{D}$. □

To ensure the existence of a finite proof tree, we need a processor which generates no children and nevertheless returns a valid result.

Definition 2.44 (Fail processor)**.** The fail processor FailP always returns (Infinite, []).

FailP is obviously correct. When implementing the techniques the described in this thesis, FailP corresponds to hitting a time or resource limit. Another trivial processor is the emptiness check of $\mathcal{S}$: It converts a trivial problem to a complexity value. It will be implicitly used after every use of another processor.

Definition 2.45 ($\mathcal{S}$-is-empty processor)**.** Let $P = (\mathrm{DT}, \mathcal{S}, \mathcal{R})$ be an CDT problem. The processors SEmptyP returns (Poly(0), []) if $\mathcal{S} = \varnothing$.

2.4. Usable Rules

For our analysis of a CDT problem $(\mathrm{DT}, \mathcal{S}, \mathcal{R})$, the TRS $\mathcal{R}$ is only relevant in so far, as the rules are needed to connect the tuples in a $(\mathrm{DT}, \mathcal{R})$-t-chain. Quite often, after the initial transformation (or a transformation like those described in Chapter 3), $\mathcal{R}$ will contain rules, which are not *usable* in this context.

Example 2.46. Recall example 2.11 and assume we have a CDT problem such that $\mathcal{R}$ consists of all the original rules

$$\mathsf{minus}(x, \mathsf{0}) \to x \tag{2.30}$$
$$\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{minus}(x, y) \tag{2.31}$$
$$\mathsf{quot}(\mathsf{0}, \mathsf{s}(y)) \to \mathsf{0} \tag{2.32}$$
$$\mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y))) \tag{2.33}$$

and Dt only consists of the following tuple:

$$\mathsf{QUOT}(\mathsf{s}(x), \mathsf{s}(y)) \to [\mathsf{QUOT}(\mathsf{minus}(x, y)), \mathsf{MINUS}(x, y)] \tag{2.34}$$

Remember that for each tuple in a t-chain we require that the left-hand side is instantiated in a way such that all its proper subterms arguments are in normal form. So, if the tuple (2.34) occurs in a t-chain, then x and y must be instantiated with normal forms. Hence, only the rules (2.30) and (2.31) can be used to rewrite the right-hand side. These two rules never produce a quot symbol, and hence the rules (2.32) and (2.33) can be never used. Therefore, they may be removed.

This concept is called *Usable Rules* and is a well-known refinement for the Dependency Pair approach. We give a semantic definition, similar to the one in [Thi07].

Definition 2.47 (Usable Rules)**.** Let $\mathcal{R}$ be a TRS, $l \to r$ a rule not necessarily in $\mathcal{R}$. Then $\mathcal{U}_\mathcal{R}(l \to r)$, the set of *Usable Rules* of $l \to r$ with regard to $\mathcal{R}$ is the minimal set such that following holds: If $l' \to r' \in \mathcal{R}$ and there exists a substitution σ such that $l\sigma$ is in normal form with regard to $\mathcal{R}$ and

$$r\sigma \xrightarrow{\mathsf{i}}{}^*_\mathcal{R} s \xrightarrow{\mathsf{i}}_{l' \to r'} t$$

is a derivation, then $l' \to r' \in \mathcal{U}_\mathcal{R}(l \to r)$. If S is a set of rules, we define $\mathcal{U}_\mathcal{R}(S) := \cup_{\rho \in S} \mathcal{U}_\mathcal{R}(\rho)$.

As this set is not computable in general, we will always work with an estimation of the usable rules

Definition 2.48 (Estimated Usable Rules)**.** A function $\mathcal{EU}$ is called an estimation of $\mathcal{U}$, if $\mathcal{U}_\mathcal{R}(\rho) \subseteq \mathcal{EU}_\mathcal{R}(\rho) \subseteq \mathcal{R}$ holds for all rules ρ and TRS $\mathcal{R}$.

For a strong estimation we refer again to [Thi07]. We will now give a first application of usable rules. Often, a CDT problem contains rules not reachable from any dependency tuple. Those rules are not relevant for the runtime complexity of the problem and can therefore be removed.

Definition 2.49 (Usable Rules Processor)**.** Let $(\text{Dt}, \mathcal{S}, \mathcal{R})$ be CDT problem and $\mathcal{EU}$ an estimation of the usable rules. Let $\mathcal{R}' := \mathcal{EU}_\mathcal{R}(\text{Dt})$. Then UsableRulesP returns $(\text{Poly}(0), \{(\text{Dt}, \mathcal{S}, \mathcal{EU}_\mathcal{R}(\mathcal{R}))\})$

Example 2.50. For the Example 2.46 given above, the processor UsableRulesP returns the CDT problem with the same set Dt and new set of rules $\mathcal{R}$

$$\begin{aligned}\mathsf{minus}(x, \mathsf{0}) &\to x\\ \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) &\to \mathsf{minus}(x, y)\end{aligned}$$

as none of the quot-rules is usable for the only tuple

$$\mathsf{QUOT}(\mathsf{s}(x), \mathsf{s}(y)) \to [\mathsf{QUOT}(\mathsf{minus}(x, y)), \mathsf{MINUS}(x, y)] \tag{2.35}$$

in this problem.

The correctness of UsableRulesP follows directly from the following lemma.

Theorem 2.51. *Let* $(\mathrm{Dt}, \mathcal{S}, \mathcal{R})$ *be CDT problem and* $\mathcal{EU}$ *an estimation of the usable rules. Then*

$$\mathrm{rc}(n, (\mathrm{Dt}, \mathcal{S}, \mathcal{R})) = \mathrm{rc}(n, \mathrm{Dt}, \mathcal{S}, \mathcal{EU}_{\mathcal{R}}(\mathrm{Dt}))$$

holds for all $n \in \mathbb{N}$.

Proof. Let $s_1 \to \overline{t_1}, s_2 \to \overline{t_2}, \dots$ be an innermost $(\mathrm{Dt}, \mathcal{R})$-t-chain with substitution σ. If a rule $l \to r$ is needed to construct this chain, then there exist $i, j \in \mathbb{N}$ such that

$$(\overline{t_i})_j\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} u \xrightarrow{\mathsf{i}}{}^*_{l\to r} v \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s_{i+}\sigma$$

holds. As $s_i\sigma$ is in normal form with regard to $\mathcal{R}$ by the definition of an innermost t-chain, $l \to r \in \mathcal{U}_{\mathcal{R}}(s_i \to \overline{t_i})$. □

2.5. Reduction Pair processor

The use of well-founded orders is one of the classic techniques for proving termination of a term rewrite system. Popular variants include simplification orders and orders based on interpretations. Those techniques do not only ensure termination, but also impose an upper bound on the complexity of the systems which can be proved terminating. As indicated in the preliminaries, most of the well-known orders do not yield polynomial bounds. A path order guaranteeing a polynomial bound on the runtime complexity is presented in [AM08].

Two notable cases of orders based on interpretations are polynomial interpretations [Lan79; CMTU05] and matrix interpretations. In the general case, polynomial interpretations impose a doubly exponential derivational complexity and a complexity of $2^{\mathcal{O}(n)}$ for linear polynomial interpretations [HL89]. Finally, [BCMT99b] shows that strong monotonicity and a restriction to the interpretation of constructor symbols guarantee a polynomial runtime complexity. Johannes Waldmann gives a short overview on the upper bounds induced by matrix interpretations in [Wal09]. In particular, upper triangular matrices of degree d induce a derivational complexity of $\mathcal{O}(n^d)$ [PSS97].

For Dependency Pairs there exists a reduction pair processor which employs reduction pairs to remove single nodes from the dependency graph. Unfortunately, this is not possible for complexity analysis. In this section, we define the variant of reduction pair processor, which can use orders to remove tuples from the $\mathcal{S}$ section. For this matter, we adapt safe reduction pairs and collapsible orders as described in [HM08a]. In addition we develop constraints a polynomial interpretation must fulfill to be usable as a reduction pair which establishes an upper polynomial bound.

First, we introduce some general notation about orders on terms [Der87; CMTU05]. A *term ordering* is a tuple $(\succsim, \succ)$ of orderings over $\mathcal{T}(\Sigma, \mathcal{V})$ for some alphabet Σ and variables $\mathcal{V}$ such that $\succsim$ is a quasi-ordering, i.e. transitive and reflexive and $\succ$ is a strict ordering, i.e., transitive and irreflexive such that $\succsim \circ \succ \subset \succ$ and $\succ \circ \succsim \subset \succ$. Here, $\circ$ denotes the composition of relations.

A relation R is *well-founded* if each sequence $t_1\ R\ t_2\ R\ t_3 \ldots$ is finite, *stable*, if it is closed under substitution, i.e., $t_1\ R\ t_2$ implies $t_1\sigma\ R\ t_2\sigma$ for all substitutions and *monotonic*, if it is closed under contexts, i.e $f(s_1, \ldots, s_n)\ R\ f(s_1, \ldots, s_{k-1}, t_k, s_{k+1}, \ldots, s_n)$ follows from $s_k\ R\ t_k$. An ordering R is *compatible* with a rewrite system $\mathcal{R}$, if $s\ R\ t$ holds for all terms with $s \to_{\mathcal{R}} t$

A term ordering $(\succsim, \succ)$ is a *reduction pair*, if $\succ$ is well-founded, $\succsim$ and $\succ$ are stable and $\succsim$ is monotonic. A reduction pair is called *safe*, if each compound symbol is strictly monotonic in all of its arguments, i.e., $\textsc{Com}(s_1, \ldots, s_n) \succ \textsc{Com}(s_1, \ldots, s_{k-1}, t_k, s_{k+1}, \ldots, s_n)$ if $s_k \succ t_k$.

We now give a definition of orders, which can be collapsed into the natural numbers. This definition is similar to the one given in [HM08a].

Definition 2.52 (Collapsible order)**.** An order $\succ$ is *G-collapsible* on a TRS $\mathcal{R}$, if there exists a function $G : \mathcal{T} \to \mathbb{N}$ such that $s \to_{\mathcal{R}} t$ and $s \succ t$ implies $G(s) > G(t)$.

So, for a term t, $G(t)$ is the maximal number of $\succ$-steps which can be performed starting with t. Hirokawa and Moser [HM08a] state that most reduction orders are collapsible.

Now we can give the definition of our reduction pair processor.

Definition 2.53 (Reduction Pair processor)**.** Let $P = (\textsc{Dt}, \mathcal{S}, \mathcal{R})$ be a CDT problem and $(\succsim, \succ)$ a safe reduction pair which is G-collapsible on $\mathcal{R} \cup \textsc{Dt}$. Let $\textsc{Dt}_{\succ} = \textsc{Dt} \cap \succ$ and $\textsc{Dt}_{\succsim} = \textsc{Dt} \cap \succsim$. Then the reduction pair processor RedPairP is defined as

$$\textsc{RedPairP}(P) := (f_{\mathsf{up}}, (\textsc{Dt}, \mathcal{S}', \mathcal{R}))$$

where $\mathcal{S}' := \mathcal{S} \setminus \textsc{Dt}_{\succ}$ if the following conditions hold:

- $\textsc{Dt}_{\succ} \cup \textsc{Dt}_{\succsim} = \textsc{Dt}$ and
- $\mathcal{R} \subseteq \succsim$ and
- there is a function f such that $f(n) \geq G(t)$ for all $t \in \mathcal{T}^{\sharp}_{B}$ of size n and $\iota(f) \leq f_{\mathsf{up}}$.

To automatize this processor, we need a reduction pair, for which the function f can be easily computed. For the reduction pair based on polynomial interpretations given in Theorem 2.56 this function is given by the interpretation function. A similar result can be derived for reduction pairs based on matrix interpretations (compare [MSW08] for how to construct a matrix order to prove the complexity of a TRS directly). Avanzini and Moser [AM09] describe a reduction pair with similar constraints for which $\iota(f)$ is bounded by $\textsc{Poly}(?)$. We give an example for the application of the reduction pair processor at the end of this section.

Lemma 2.54 (Correctness of RedPairP)**.** *The processor* RedPairP *is correct.*

Proof. Let $s \in \mathcal{T}_B^\sharp$ be a basic term. All terms in an innermost $(\mathrm{DT} \cup \mathcal{R})$-derivation are well-formed. As the compound symbols are strictly monotonic, all DT-steps in such a derivation take place at a monotonic position.

So if $t \xrightarrow{\mathsf{i}}_{\mathrm{DT}_\succ} u$ occurs in a derivation of s, then $t \in \mathcal{T}_T^\sharp$ and therefore $t \succ u$. On the other hand, $\succsim$ is monotonic too, so for all $t, u \in \mathcal{T}_B$ we have $t \succsim u$ if $t \to_{\mathcal{R} \cup \mathrm{DT}_\succsim} u$.

Now, let

$$s = s_0 \xrightarrow{\mathsf{i}}_{\nu_0} t_0 \xrightarrow{\mathsf{i}}_{\mathcal{R}}^* s_1 \xrightarrow{\mathsf{i}}_{\nu_1} t_1 \xrightarrow{\mathsf{i}}_{\mathcal{R}}^* s_2 \cdots$$

be a $\to_{\mathrm{DT}/\mathcal{R}}$-derivation, where $\nu_i \in \mathrm{DT}$ -for all i. Then

$$s = s_0 \mathbin{\succcurlyeq_0} t_0 \succsim^* s_1 \mathbin{\succcurlyeq_1} t_1 \succsim^* s_2 \cdots .$$

holds. Here $\succcurlyeq_i$ is $\succ$ if $\nu_i \in \mathcal{S}$ and $\succsim$ else. Let $I = i_1 < i_2, \ldots$ be the sequence of indexes with $\succcurlyeq_i = \succ$. For each $i \in I$ holds $s_i \succ t_i$. As $\succsim \cdot \succ \subseteq \succ$ and $\succ \cdot \succsim \subseteq \succ$, we get $s \succ t_{i_1} \succ t_{i_2} \succ \cdots$ and therefore $G(s) > G(t_{i_1}) > G(t_{i_2}) > \cdots$.

Hence $G(s)$ is an upper bound of the number of $\mathrm{DT}_\succ$-steps in any innermost $(\mathrm{DT} \cup \mathcal{R})$-derivation of s. By Remark 2.24 $|\mathrm{MCT}(s)|_{\mathcal{S}}$ equals the maximal number of $\mathcal{S}$-steps in a $\mathrm{DT} \cup \mathcal{R}$-derivation, so this gives an upper bound for $|\mathrm{MCT}(T)|_{\mathrm{DT}_\succ}$, i.e., $\mathrm{rc}_{\mathfrak{D}}(\mathrm{DT}, \mathrm{DT}_\succ, \mathcal{R}) \leq f_{\mathsf{up}}$.

After we have established this, we have to show that

$$\mathrm{rc}_{\mathfrak{D}}(P) \leq f_{\mathsf{up}} + (\mathrm{rc}_{\mathfrak{D}}(\mathrm{DT}, \mathcal{S}'\mathcal{R}) \ominus \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(\mathrm{DT}, \mathcal{S}'\mathcal{R})) + \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P)$$

holds. But this holds, as

$$\begin{aligned}
f_{\mathsf{up}} + \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P) &\geq \mathrm{rc}_{\mathfrak{D}}(\mathrm{DT}, \mathrm{DT}_\succ, \mathcal{R}) + \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P) \\
&= \mathrm{rc}_{\mathfrak{D}}(\mathrm{DT}, \mathrm{DT}_\succ, \mathcal{R}) + \mathrm{rc}_{\mathfrak{D}}(\mathrm{DT}, \mathrm{DT} \setminus \mathcal{S}, \mathcal{R}) \\
&= \mathrm{rc}_{\mathfrak{D}}(\mathrm{DT}, \mathrm{DT}_\succ \cup (\mathrm{DT} \setminus \mathcal{S}), \mathcal{R}) \\
&= \mathrm{rc}_{\mathfrak{D}}(\mathrm{DT}, \mathrm{DT} \setminus (\mathcal{S} \setminus \mathrm{DT}_\succ), \mathcal{R}) \\
&= \mathrm{rc}_{\mathfrak{D}}(\mathrm{DT}, \mathrm{DT} \setminus \mathcal{S}', \mathcal{R}) \\
&= \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(\mathrm{DT}, \mathcal{S}', \mathcal{R})
\end{aligned}$$

is true, so if $\mathrm{rc}_{\mathfrak{D}}(\mathrm{DT}, \mathcal{S}', \mathcal{R}) \leq \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(\mathrm{DT}, \mathcal{S}', \mathcal{R})$, then also $\mathrm{rc}_{\mathfrak{D}}(\mathrm{DT}, \mathcal{S}', \mathcal{R}) \leq f_{\mathsf{up}} + \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P)$. □

An alternative to the reduction pair processor would be a technique which applies an order to solve the whole problem at once. But the big benefit of the reduction pair processor is that the constraints for the order are much weaker. If we want to prove the complexity of a CDT problem with a single order, we have to orient all tuples strictly (and the rules weakly). Using the reduction pair processor it suffices to orient one tuple strictly. For some problems this is crucial to the success of a proof. We will see an example for that after we introduce orders based on polynomial interpretations.

But there are further advantages to the reduction pair approach. By the modular nature of this processor, it is possible to use different types of orders to simplify the problem iteratively. But not only the proof of the complexity of a problem is modularized, but also the theory needed to find applicable orders. For orders like polynomial or matrix interpretations it is well-known how to construct the constraints necessary to fulfill the requirements of the reduction pair processor, so their adaption is a simple task. In particular there is an extension of the Reduction Pair processor which employs usable rules w.r.t argument filtering systems [GTSF06]: This extension allows ignoring some of the usable rules of a system under certain conditions. If we want to use this technique for our complexity analysis, we only have to redo the correctness proof for the reduction pair processor, not for correctness proof for the orders we employ.

To implement this processor we must be able to (automatically) find a reduction pair. Polynomial interpretations are a well-understood and easily automated mechanism for finding reduction pairs and obviously collapsible. Only few additional constraints are necessary to use them for proving polynomial runtime complexity. First, we recall the short definition of an interpretation and the associated ordering.

Definition 2.55 (Interpretation)**.** Let $(\mathcal{A}, \succ_{\mathcal{A}}, \succsim_{\mathcal{A}})$ be an algebra. Let Σ be a signature and $\mathcal{V}$ a probably infinite set of variables. For each n-ary function symbol $f \in \Sigma$ let $[f]$ be a n-ary function $\mathcal{A} \times \cdots \times \mathcal{A} \to \mathcal{A}$. $[\cdot]$ is called an *$\mathcal{A}$-interpretation.* A *(variable) assignment* is a mapping $\mathcal{V} \to \mathbb{N}$.

We extend $[\cdot]$ to $\mathcal{T}(\Sigma, \mathcal{V})$ by setting

$$[t]_\alpha := \begin{cases} \alpha(x) & \text{if } t = x \in \mathcal{V} \\ [f]_\alpha([t_1]_\alpha, \ldots, [t_n]_\alpha) & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

So $[t]$ is a function mapping an assignment to $\mathcal{A}$. For terms s, t, we set $s \succ t$ if and only if $[s]_\alpha \succ_A [t]_\alpha$ for all assignments α. Analogous, $s \succsim t$ if and only if $[s]_\alpha \succsim_A [t]_\alpha$.

We say a rule $l \to r$ is strictly decreasing, if $l \succ r$ and weakly decreasing, if $l \succsim r$.

In this section, we are interested primarily in interpretations over the algebra $(\mathbb{N}, >, \geq)$, where $>$ and $\geq$ are the usual greater and greater-equal relations on the natural numbers.

A simple idea to derive a complexity bound is the following: Let $[\cdot]$ be an interpretation over $\mathbb{N}$. If $s \to t$ implies $[s] > [t]$, i.e. the order associated with $[\cdot]$ is collapsible, then obviously the derivation length of s with regard to $\to$ is smaller or equal then $[s]$. Bonfante, Cichon, Marion, and Touzet [BCMT99b] already outlined the restrictions needed for a polynomial order to fulfill this requirement. But for reduction pairs, those restrictions can be loosened considerably.

Theorem 2.56. *Let $\mathcal{R}, \mathcal{P}$ be TRSs over Σ and $\mathcal{V}$. Let $[\cdot]$ be a polynomial interpretation such that the associated ordering $(\succsim, \succ)$ is a reduction pair with $\mathcal{R} \subseteq \succsim$ and $\mathcal{P} \subseteq \succ$. If for each n-ary compound symbol* Com *the interpretation is $[\textsc{Com}](x_1, \ldots, x_n) = x_1 + \ldots + x_n$, the reduction pair is a safe reduction pair.*

If in addition $[c](x_1, \ldots, x_n) = a_1 x_1 + \cdots + a_n x_n + b$ with $a_i \in \{0, 1\}$ and $b \in \mathbb{N}$ for all n-ary constructor symbols $f \in \mathcal{C}_\mathcal{R}$, then $[f](k)$ is an upper bound for the number of $\mathcal{P}$-steps in a derivation of a term with size k and root symbol f.

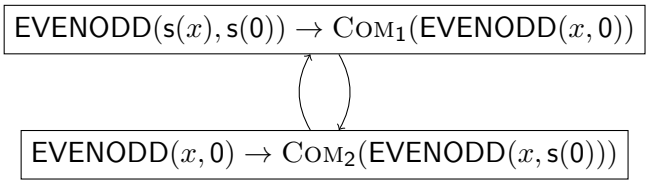
Proof. The first restriction is obvious. The second restriction ensures that the interpretation of a constructor term is at most its size k (multiplied with a factor c independent of k). So we get also

$$c' \cdot [f](m_1, \ldots, m_n) \geq [f](cm_1, \ldots, cm_n) \geq \mathrm{dl}(f(t_1, \ldots, t_n), \to_{\mathrm{DT}/\mathcal{R}})$$

for all defined symbols f and constructor terms $t_1, \ldots, t_n$ such that $|t_i| \leq m_i$. □

There are well-established techniques to find reduction pairs with the help of polynomial interpretations [GTSF06] [CMTU05] [FGMSTZ07]. Those techniques allow giving arbitrary restrictions to the form of the constructed polynomials, so they can easily be employed to automate the above theorem.

Example 2.57. Consider the CDT problem $(\mathrm{DT}, \mathcal{S}, \mathcal{R})$ described by the following dependency graph:



These two tuples cannot be oriented strictly by a polynomial interpretation. The second rule requires $[0] > [\mathsf{s}(0)]$ and hence $[s]$ must be a constant as all coefficients are natural numbers. But then $[\mathsf{s}(x)] > [x]$ does not hold for all variable assignments, so we need $[0] < [\mathsf{s}(0)]$, too.

But orienting a single tuple strictly (and the other weakly) is easy. The interpretation

$$[\mathsf{EVENODD}](x, y) = x \qquad [\mathsf{s}](x) = x + 1 \qquad [0] = 0$$

orients the first tuple strict and hence it may be removed from $\mathcal{S}$. By applying the KnownnessP processor presented in the next section, we may remove the other tuple to and hence have proved linear complexity for this problem.

3. Complexity Dependency Graph

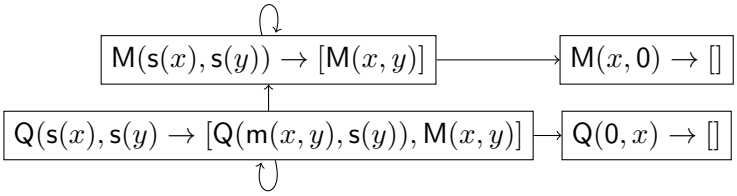
The Dependency Graph is an important part of the original Dependency Pair approach. The idea is to expose which pairs can follow each other in a chain. For dependency tuples, we use essentially the same definition.

Definition 3.1 (Complexity Dependency Graph)**.** For a CDT problem $(\mathrm{D}\mathrm{T}, \mathcal{S}, \mathcal{R})$, the *Complexity Dependency Graph* is the directed graph $G = (V, E)$ with nodes $V = \mathrm{D}\mathrm{T}$ and two nodes $v_1 = s_1 \to \overline{t_1}$, $v_2 = s_2 \to \overline{t_2}$ are connected in G, i.e., $(v_1, v_2) \in E$ iff $s_1 \to \overline{t_1}$, $s_2 \to \overline{t_2}$ is a t-chain.

We say an edge (v_1, v_2) *exists for the rhs* i, iff $v_1 \langle i \rangle v_2$ holds.

We say that u is a *predecessor* (*successor*) of v, if $(u, v) \in E$ ($(v, u) \in E$). A *path* is a sequence of nodes $u_1, u_2, \ldots$ such that u_{i+1} is a successor of u_i for all $i > 1$. A node v is *reachable* from u, if there is a (probably non-empty) path from v to u.

Example 3.2. The Complexity Dependency Graph for Example 2.11 is the following graph (we shortened the function symbols to one character):



In termination analysis, it is sound (and complete) to decompose $\mathcal{P}$ into the strongly connected components of the graph. Here, we follow the usual convention for dependency pairs and require that an SCC contains at least one edge. This differs from the usual graph-theoretic definition, where a single, unconnected node is also considered a SCC.

Definition 3.3 (Strongly Connected Component)**.** A subset C of a graph G is *strongly connected*, if for every pair of nodes $u, v \in C$ there are a non-empty paths from u to v and from v to u. Such a set is called a *strongly connected component* (or *SCC*), if it is not a proper subset of another strongly connected set.

Definition 3.4 (Mathematical SCC)**.** A mathematical SCC (MSCC) is an SCC or a single node not part of any SCC.

Sometimes, we need a more fine grained look on SCCs.

Definition 3.5 (Cycle)**.** A subset $C = \{c_1, \ldots, c_n\}$ of a graph G is called a *cycle*, if the c_i are pairwise disjoint and $c_1, c_2, \ldots, c_n, c_1$ is a path in G.

Obviously, each cycle is contained in a SCC. Unfortunately, the SCC-decomposition of the Complexity Dependency Graph is not complexity preserving.

Example 3.6. Consider the following system computing powers of two.

$$\mathsf{double}(\mathsf{0}) \to \mathsf{0} \tag{3.1}$$
$$\mathsf{double}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{double}(x))) \tag{3.2}$$
$$\mathsf{exp}(\mathsf{0}) \to \mathsf{s}(\mathsf{0}) \tag{3.3}$$
$$\mathsf{exp}(\mathsf{s}(x)) \to \mathsf{double}(\mathsf{exp}(x)) \tag{3.4}$$

As we have seen in Example 2.5, the derivation length of $\mathsf{double}^n(x)$ is exponential in n. Evaluating $\mathsf{exp}(\mathsf{s}^n(x))$ yields $\mathsf{double}^n(x)$, so the system has indeed exponential runtime complexity. The CDT-transformation yields the following pairs

$$\mathsf{DOUBLE}(\mathsf{0}) \to [] \tag{3.5}$$
$$\mathsf{DOUBLE}(\mathsf{s}(x)) \to [\mathsf{DOUBLE}(x)] \tag{3.6}$$
$$\mathsf{EXP}(\mathsf{0}) \to [] \tag{3.7}$$
$$\mathsf{EXP}(\mathsf{s}(x)) \to [\mathsf{EXP}(x), \mathsf{DOUBLE}(\mathsf{exp}(x))] \tag{3.8}$$

and {3.6} and {3.8} are the SCCs of the graph. It is easy to see that the runtime complexity of both SCCs is linear. By analyzing each SCC on its own, we ignore the fact that 3.8 calls 3.6 with arguments exponential in the size of the start term.

As one of the main advantages of the Dependency Graph is the ability to decompose the original problem into smaller ones, we need to find a replacement for the SCC-decomposition. An obviously correct, but much weaker idea is to decompose the graph into its components.

Definition 3.7 (Component)**.** A *component* of a graph G is an inclusion-maximal subset $C \subseteq G$ such that for all pairs of nodes $u, v \in C$ there is a path from u to v in the underlying undirected graph.

Theorem 3.8 (Graph Split)**.** *Let* $(\mathrm{D_T}, \mathcal{S}, \mathcal{R})$ *be a CDT problem and* $C_1, \ldots, C_k$ *the components of the dependency graph. Then we have*

$$\min_{1 \le i \le k} \mathrm{rc}(n, (C_i, \mathcal{S} \cap C_i, \mathcal{R})) \le \mathrm{rc}(n, (\mathrm{D_T}, \mathcal{S}, \mathcal{R})) \le \max_{1 \le i \le k} \mathrm{rc}(n, (C_i, \mathcal{S} \cap C_i, \mathcal{R})).$$

Proof. By the definition of the dependency graph, a t-chain starting with a node $s \to t$ can only reach nodes in the same component. Therefore all t-chains in a chain tree lie in only one component and therefore for each $(\mathrm{D_T}, \mathcal{R})$-chain tree T there exists an $1 \le i \le k$ such that T is an $(C_i, \mathcal{R})$-chain tree. □

To compute the Graph-Split, we need the dependency graph. Unfortunately, this graph is not computable in general. But for all our needs an approximation containing at least all the edges of the dependency graph suffices. As our definition of the Complexity Dependency Graph is essentially the same as the definition of the Dependency Graph for termination, we can easily use well-known graph approximations [AG00; Mid01; GTS05a; Thi07] as described in [Thi07].

3.1. Simplifying $\mathcal{S}$

In this section we will see how to use the DT versus $\mathcal{S}$ distinction to simplify proofs of complexity. This distinction resembles the notion of relative rewriting [BD86]: We use both nodes in $\mathcal{S}$ and $\mathrm{DT} \setminus \mathcal{S}$, but count only nodes in $\mathcal{S}$. So we when we remove a tuple from $\mathcal{S}$, we call this *relative tuple removal*.

While the techniques presented here do not make the graph by itself simpler, they reduce the number of pairs which have to be oriented strictly by the reduction pair processor. First, we make a simple observation.

Remark 3.9 (Relative Graph Split). Let $(\mathrm{DT}, \mathcal{S}, \mathcal{R})$ be a CDT problem and $\mathcal{S}_1 \cup \cdots \cup \mathcal{S}_k$ be a partition of $\mathcal{S}$. Then

$$\mathrm{rc}(n, (\mathrm{DT}, \mathcal{S}, \mathcal{R})) = \sum_{1 \leq i \leq k} \mathrm{rc}(n, (\mathrm{DT}, \mathcal{S}_i, \mathcal{R}))$$

holds for all $n \in \mathbb{N}$.

This remark allows us to prove upper bounds for each tuple separately: If we we know the upper bound for some of the tuples, we may remove them from $\mathcal{S}$. As Example 2.39 demonstrates, the naive way of turning Remark 3.9 into a processor is incorrect. We will see how to use this correctly later.

Often, we can derive an upper bound for the occurrences of one node, if we know how often its predecessors can occur.

Lemma 3.10 (Knownness propagation)**.** *Let $(\mathrm{DT}, \mathcal{S}, \mathcal{R})$ be a CDT problem with dependency graph (DT, E) and $\nu \in \mathrm{DT}$. Let N be the set of incoming nodes of ν, i.e., $(\rho, \nu) \in E$ for all $\rho \in N$. Furthermore, let k be the maximal number of right-hand sides of a tuple in N. If $|T|_N = s$ for an arbitrary $(\mathrm{DT}, \mathcal{S}, \mathcal{R})$-chain tree T, then $|T|_{\{\rho\}} \leq 1 + ks$.*

Proof. Let T be a $(\mathrm{DT}, \mathcal{S}, \mathcal{R})$-chain tree and v an occurrence of ν in T. Then either v is the root node, or v is successor of a node in N. Each of these nodes has at most k right-hand sides, so $|T|_{\{\nu\}} \leq 1 + ks$ follows. □

We can use this theorem to reduce the work needed to be done by the reduction pair processor.

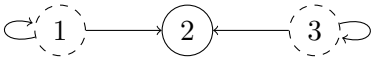
Definition 3.11 (Knownness propagation processor)**.** Let $P = (\mathrm{DT}, \mathcal{S}, \mathcal{R})$ be a CDT problem. Let $\nu \in \mathrm{DT}$ be a node in the dependency graph and N be the set of incoming nodes of ν. If $N \cap \mathcal{S} = \varnothing$, then the *knownness propagation processor* KnownnessP returns

$$\textsc{KnownnessP}(P) := (\textsc{Poly}(0), (\mathrm{DT}, \mathcal{S} \setminus \{\nu\}, \mathcal{R}))$$

Lemma 3.12 (Correctness of KnownnessP)**.** *The processor* KnownnessP *is correct.*

Proof. Follows from Lemma 3.10, as $\mathrm{rc}^{\mathrm{c}}_{\mathfrak{D}}(\mathrm{DT}, \mathcal{S} \setminus \{\nu\}, \mathcal{R}) = \mathrm{rc}^{\mathrm{c}}_{\mathfrak{D}}(P)$. □

Example 3.13. Recall the graph of Example 2.36:



As none of the incoming nodes of node (2) is in $\mathcal{S}$, the KnownnessP processor returns the trivial problem

which can be solved by the SEmptyP processor.

In the next section we will see that some nodes in $\mathrm{D_T} \setminus \mathcal{S}$ can even be removed completely from the problem. Combining this with the results above, we can finally get some kind of SCC processor.

3.2. Graph simplification

In this section we build on the results of the (relative) graph split and try to make components as SCC-like as possible. Of particular interest are those nodes, which can only occur at the beginning or at the end of a t-chain, i.e., those nodes which have no incoming respective outgoing edge in the dependency graph. We call those nodes *leading* respective *trailing* (or *dangling*, if we mean both).

We have demonstrated in Example 3.6 that the SCC-split is unsound as we cannot see anymore that an SCC was reached with arguments considerably bigger then the start term. This observation suggests that paths not leading to a SCC may be removed safely.

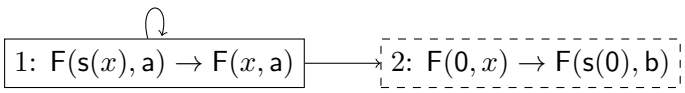
Theorem 3.14 (Removal of trailing nodes)**.** *Let* $(\mathrm{D_T}, \mathcal{S}, \mathcal{R})$ *be a CDT problem with dependency graph* G*. Let* $V \subseteq \mathrm{D_T} \setminus \mathcal{S}$ *be a set containing all nodes reachable from* V*. Then*

$$\mathrm{rc}(n, (\mathrm{D_T}, \mathcal{S}, \mathcal{R})) = \mathrm{rc}(n, \mathrm{D_T}', \mathcal{S}, \mathcal{R})$$

holds, where $\mathrm{D_T}' := \mathrm{D_T} \setminus V$*.*

Proof. As all nodes reachable from V are contained in V, there can be no node from $\mathrm{D_T} \setminus V$ after a node from V on an innermost $(\mathrm{D_T}, \mathcal{S}, \mathcal{R})$-t-chain. Hence, removing all V-nodes from a $(\mathrm{D_T}, \mathcal{S}, \mathcal{R})$-chain tree T still yields a valid $(\mathrm{D_T}', \mathcal{S}, \mathcal{R})$-chain tree T'. As V and $\mathcal{S}$ are disjoint, $|T|_{\mathcal{S}} = |T'|_{\mathcal{S}}$ holds. □

Example 3.15. Consider the CDT problem given by the following dependency graph.



It is not possible to orient tuple (1) strictly and tuple (2) weakly with an polynomial order. Tuple (1) requires that $[s](x) > x$ for all $x \in \mathbb{N}$, while the second tuple requires that $[0] \leq [s(0)]$. By applying Theorem 3.14, node (2) can be removed and the remaining tuple can easily be oriented strictly.

Note that the above theorem even allows us to remove whole SCCs, if they are marked as being of known complexity. We have restricted the theorem to known nodes, but we can use a combination of Remark 3.9, Lemma 3.10 and Lemma 3.21 to delete arbitrary trailing nodes. Now, we finally are able to find a worthy equivalent to the Dependency Graph Processor which allows splitting a DP problem in its SCCs.

Definition 3.16 (Preceding MSCCs)**.** Let G be a graph, $C_1 \neq C_2$ two mathematical SCCs of G. If there is an edge (c_1, c_2) for some $c_1 \in C_1$ and $c_2 \in C_2$, we say C_1 *precedes* C_2 and write $C_1 \lhd C_2$. We denote by $\lhd^*$ the transitive and reflexive closure of $\lhd$ and by $\lhd^+$ the transitive closure.

By ${C_1}^{\lhd} := \bigcup_{C \lhd^* C_1} C$ we denote the *reachability closure* of C_1, i.e., the set of all nodes, from which C_1 can be reached.

Definition 3.17 (SCC Split processor)**.** Let $P = (\textsc{Dt}, \mathcal{S}, \mathcal{R})$ be a CDT problem and $C_1, \ldots, C_k$ the SCCs of the dependency graph. For each SCC C_i let $\textsc{Dt}_i := {C_i}^{\lhd}$. Then the SCC Split processor SccSplitP returns

$$\textsc{SccSplitP}(P) = (\textsc{Poly}(0), [(\textsc{Dt}_1, C_1 \cap \mathcal{S}, \mathcal{R}), \ldots, (\textsc{Dt}_k, C_k \cap \mathcal{S}, \mathcal{R})])$$

Proving the correctness of SccSplitP turns out to be rather technical. We will give the idea of the proof directly and move the technical details to auxiliary lemmas.

Lemma 3.18 (Correctness of SccSplitP)**.** *The processor* SccSplitP *is correct.*

Proof. Let G be the dependency graph of P and $\mathfrak{C}$ be the set of mathematical SCCs of G. For a $C \in \mathfrak{C}$, let $P_C := ({C}^{\lhd}, C \cap \mathcal{S}, \mathcal{R})$. By remark 3.9 we have

$$\begin{aligned}
\mathrm{rc}_{\mathfrak{D}}(P) &\leq \sum_{C \in \mathfrak{C}} \mathrm{rc}_{\mathfrak{D}}(P_C) \\
&\leq \sum_{C \in \mathfrak{C}} \mathrm{rc}_{\mathfrak{D}}({C}^{\lhd}, {C}^{\lhd} \cap S, \mathcal{R}) \\
&\leq \sum_{C \in \mathfrak{C}^T} \mathrm{rc}_{\mathfrak{D}}({C}^{\lhd}, {C}^{\lhd} \cap S, \mathcal{R})
\end{aligned}$$

where $\mathfrak{C}^T$ is the set of MSCCs which have no successor (i.e., not $C \lhd C'$ for any pair of $C \in \mathfrak{C}^T$ and $C' \in \mathfrak{C}$). In Lemma 3.19 we show that

$$\mathrm{rc}_{\mathfrak{D}}({C}^{\lhd}, {C}^{\lhd} \cap \mathcal{S}, \mathcal{R}) \leq \sum_{B \lhd^* C} (\mathrm{rc}_{\mathfrak{D}}(P_B) \ominus \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_B)) + \mathrm{rc}_{\mathfrak{D}}(P)$$

holds for all $C \in \mathfrak{C}$. Together with the above, this proves

$$\mathrm{rc}_{\mathfrak{D}}(P) \leq \sum_{B \lhd^* C} (\mathrm{rc}_{\mathfrak{D}}(P_B) \ominus \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_B)) + \mathrm{rc}_{\mathfrak{D}}(P).$$

In addition, Lemma 3.20 tells us that $\mathrm{rc}_{\mathfrak{D}}(P_C) \ominus \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_C) = \textsc{Poly}(0)$ for all $C \in \mathfrak{C}$ which are only MSCCs but no SCCs. So we finally arrive at

$$\mathrm{rc}_{\mathfrak{D}}(P) \leq \sum_{1 \leq i \leq k} (\mathrm{rc}_{\mathfrak{D}}(P_{C_i}) \ominus \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_{C_i})) + \mathrm{rc}_{\mathfrak{D}}(P),$$

which proves the correctness of the processor. □

The statement of the next lemma is the following: All nodes marked as known in P_C were either already known in P or there is preceding MSCC where this nodes are marked as unknown. As $\lhd^+$ is a well-founded order, this guarantees that we have no cyclic dependencies which caused Example 2.39 to be incorrect.

Lemma 3.19. *In notation of Lemma 3.18,*

$$\mathrm{rc}_{\mathfrak{D}}(C^{\lhd}, C^{\lhd} \cap \mathcal{S}, \mathcal{R}) \leq \sum_{B \lhd^* C} (\mathrm{rc}_{\mathfrak{D}}(P_B) \ominus \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_B)) + \mathrm{rc}_{\mathfrak{D}}(P)$$

holds for all $C \in \mathfrak{C}$.

Proof. The *depth* of a $C \in \mathfrak{C}$ is the maximal number of $\lhd$-steps in a sequence $A \lhd \ldots \lhd C$. We will do a parallel induction on two propositions:

(1) Proposition $\Phi(n)$:

$$\mathrm{rc}_{\mathfrak{D}}(C^{\lhd}, C^{\lhd} \cap \mathcal{S}, \mathcal{R}) \leq \sum_{B \lhd^* C} (\mathrm{rc}_{\mathfrak{D}}(P_B) \ominus \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_B)) + \mathrm{rc}_{\mathfrak{D}}(P)$$

holds for all $C \in \mathfrak{C}$ with $\mathrm{depth}(C) = n$.

(2) Proposition $\Psi(n)$:

$$\mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_C) \leq \sum_{B \lhd^+ C} (\mathrm{rc}_{\mathfrak{D}}(P_B) \ominus \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_B)) + \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P)$$

holds for all $C \in \mathfrak{C}$ with $\mathrm{depth}(C) = n$.

We will use the following induction hypothesis: If $\Phi(k)$ holds all $k < n$, then $\Psi(n)$ holds. Also, if $\Psi(n)$ and $\Phi(k)$ hold for all $k < n$, then $\Psi(n)$ holds too.

Proof for Φ. If $\mathrm{depth}(C) = 0$, then

$$\mathrm{rc}_{\mathfrak{D}}(C^{\lhd}, C^{\lhd} \cap \mathcal{S}, \mathcal{R}) = \mathrm{rc}_{\mathfrak{D}}(C^{\lhd}, C \cap \mathcal{S}) = \mathrm{rc}_{\mathfrak{D}}(P_C)$$

and

$$\begin{aligned} \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_C) &= \mathrm{rc}_{\mathfrak{D}}(C^{\lhd}, C^{\lhd} \setminus \{C \cap \mathcal{S}\}, \mathcal{R}) = \mathrm{rc}_{\mathfrak{D}}(C^{\lhd}, C^{\lhd} \setminus \mathcal{S}, \mathcal{R}) \\ &\leq \mathrm{rc}_{\mathfrak{D}}(\mathrm{D}\mathrm{T}, \mathrm{D}\mathrm{T} \setminus \mathcal{S}, \mathcal{R}) = \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P) \end{aligned}$$

hold and hence

$$\mathrm{rc}_{\mathfrak{D}}(C^{\lhd}, C^{\lhd} \cap \mathcal{S}, \mathcal{R}) \leq (\mathrm{rc}_{\mathfrak{D}}(P_C) \ominus \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_c)) + \mathrm{rc}_{\mathfrak{D}}(P)$$

is true. Now we handle the case $\operatorname{depth}(C) = n$:

$$\begin{aligned}
&\operatorname{rc}_{\mathfrak{D}}(C^{\lhd}, C^{\lhd} \cap \mathcal{S}, \mathcal{R}) \\
&= \operatorname{rc}_{\mathfrak{D}}(C^{\lhd}, C \cap \mathcal{S}, \mathcal{R}) + \operatorname{rc}_{\mathfrak{D}}(C^{\lhd}, (C^{\lhd} \setminus C) \cap \mathcal{S}, \mathcal{R})) \\
&= \operatorname{rc}_{\mathfrak{D}}(P_C) + \operatorname{rc}_{\mathfrak{D}}(C^{\lhd} \setminus C, (C^{\lhd} \setminus C) \cap \mathcal{S}, \mathcal{R})) && \text{(by Thm 3.14)} \\
&\leq \operatorname{rc}_{\mathfrak{D}}(P_C) + \sum_{B \lhd^+ C} \operatorname{rc}_{\mathfrak{D}}(B^{\lhd}, B^{\lhd} \cap \mathcal{S}, \mathcal{R}) \\
&= \operatorname{rc}_{\mathfrak{D}}(P_C) + \sum_{B \lhd^+ C} \sum_{A \lhd^* C} (\operatorname{rc}_{\mathfrak{D}}(P_A) \ominus \operatorname{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_A)) + \operatorname{rc}^{\mathsf{c}}_{\mathfrak{D}}(P) && \text{(IH } \Phi(n-1)) \\
&= \operatorname{rc}_{\mathfrak{D}}(P_C) + \sum_{B \lhd^+ C} ((\operatorname{rc}_{\mathfrak{D}}(P_B) \ominus \operatorname{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_B)) + \operatorname{rc}^{\mathsf{c}}_{\mathfrak{D}}(P)) \\
&= (\operatorname{rc}_{\mathfrak{D}}(P_C) \ominus \operatorname{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_C)) + \sum_{B \lhd^+ C} (\operatorname{rc}_{\mathfrak{D}}(P_B) \ominus \operatorname{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_B)) + \operatorname{rc}^{\mathsf{c}}_{\mathfrak{D}}(P) && \text{(IH } \Psi(n)) \\
&= \sum_{B \lhd^* C} (\operatorname{rc}_{\mathfrak{D}}(P_B) \ominus \operatorname{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_B)) + \operatorname{rc}^{\mathsf{c}}_{\mathfrak{D}}(P)
\end{aligned}$$

Proof for Ψ. If $\operatorname{depth}(C) = 0$, then

$$\begin{aligned}
\operatorname{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_C) &= \operatorname{rc}^{\mathsf{c}}_{\mathfrak{D}}(C^{\lhd}, C \cap \mathcal{S}, \mathcal{R}) = \operatorname{rc}^{\mathsf{c}}_{\mathfrak{D}}(C, C \cap \mathcal{S}, \mathcal{R}) \\
&= \operatorname{rc}_{\mathfrak{D}}(C, C \setminus \mathcal{S}, \mathcal{R}) = \operatorname{rc}_{\mathfrak{D}}(\mathrm{D}\mathrm{T}, C \setminus \mathcal{S}, \mathcal{R}) \\
&\leq \operatorname{rc}\mathrm{D}\mathrm{T}, \mathrm{D}\mathrm{T} \setminus \mathcal{S}, \mathcal{R} = \operatorname{rc}^{\mathsf{c}}_{\mathfrak{D}}(P)
\end{aligned}$$

holds. Now the induction step for $\operatorname{depth}(C) = n$. We have

$$C^{\lhd} \setminus (\mathcal{S} \cap C) = \bigcup_{B \lhd^+ C} B \cup (C \setminus \mathcal{S})$$

and hence

$$\begin{aligned}
\operatorname{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_C) &= \sum_{B \lhd^+ C} \operatorname{rc}_{\mathfrak{D}}(C^{\lhd}, B \cap \mathcal{S}, \mathcal{R}) + \operatorname{rc}_{\mathfrak{D}}(C^{\lhd}, C^{\lhd} \setminus \mathcal{S}, \mathcal{R}) \\
&= \sum_{B \lhd^+ C} \operatorname{rc}_{\mathfrak{D}}(P_B) + \operatorname{rc}_{\mathfrak{D}}(C^{\lhd}, C \setminus \mathcal{S}, \mathcal{R}) && \text{(by Thm 3.14)} \\
&\leq \sum_{B \lhd^+ C} \operatorname{rc}_{\mathfrak{D}}(P_B) + \operatorname{rc}_{\mathfrak{D}}(\mathrm{D}\mathrm{T}, \mathrm{D}\mathrm{T} \setminus \mathcal{S}, \mathcal{R}) \\
&\leq \sum_{B \lhd^+ C} \sum_{A \lhd^* B} ((\operatorname{rc}_{\mathfrak{D}}(P_A) + \operatorname{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_A)) + \operatorname{rc}^{\mathsf{c}}_{\mathfrak{D}}(P)) + \operatorname{rc}_{\mathfrak{D}}(P) && \text{(IH } \Phi(n-1)) \\
&\leq \sum_{B \lhd^+ C} \sum_{A \lhd^* B} (\operatorname{rc}_{\mathfrak{D}}(P_A) + \operatorname{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_A)) + \operatorname{rc}_{\mathfrak{D}}(P) \\
&\leq \sum_{B \lhd^* C} (\operatorname{rc}_{\mathfrak{D}}(P_B) + \operatorname{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_B)) + \operatorname{rc}_{\mathfrak{D}}(P).
\end{aligned}$$

This proves the lemma. □

Lemma 3.20. *Let C be a single node not part of any SCC. Then $\operatorname{rc}_{\mathfrak{D}}(P_C) \leq \operatorname{rc}^{\mathsf{c}}_{\mathfrak{D}}(P_C)$ in the notation of Lemma 3.18.*

Proof. Follows directly from Lemma 3.21 below. □

If we consider nodes not in any SCC, we observe that they do not heavily influence the asymptotic complexity; how often such a node can occur in a chain tree is determined by the SCCs from which this node is reachable.

Lemma 3.21. *Let $P = (\mathrm{Dt}, \mathcal{S}, \mathcal{R})$ be a CDT problem such that no $s \in ST$ occurs in any SCC of the dependency graph. Then there exists constants c_1, c_2 such that*

$$\mathrm{rc}(n, (\mathrm{Dt}, \mathcal{S}, \mathcal{R})) \leq c_1 + c_2 \cdot \mathrm{rc}^{\mathsf{c}}(n, (\mathrm{Dt}, \mathcal{S}, \mathcal{R}))$$

holds for all $n \in \mathbb{N}$.

Proof. Induction over $|\mathcal{S}|$. If $|\mathcal{S}| = 0$, we have $\mathrm{rc}(n, (\mathrm{Dt}, \mathcal{S}, \mathcal{R})) = 0$ and hence this case holds. Else, as no $s \in \mathcal{S}$ is part of any SCC, there exists a $s \in \mathcal{S}$ such that $\mathcal{S} \cap \mathrm{Pred} = \varnothing$, where Pred is the set of predecessors of s (i.e., the complexity of all predecessors is known). Set $\mathcal{S}' := \mathcal{S} \setminus \{s\}$ and let k be the maximal number of right-hand sides in Dt. Then, by using knownness propagation (3.10), we have

$$\mathrm{rc}(n, (\mathrm{Dt}, \{s\}, \mathcal{R})) \leq 1 + k \cdot \mathrm{rc}(n, (\mathrm{Dt}, \mathrm{Pred}, \mathcal{R})) \leq 1 + k \cdot \mathrm{rc}^{\mathsf{c}}(n, P)$$

as $P \subseteq \mathrm{Dt} \setminus \mathcal{S}$. Hence

$$\begin{aligned}
\mathrm{rc}(n, P) &\leq 1 + \mathrm{rc}(n, (\mathrm{Dt}, \{s\}, \mathcal{R})) + \mathrm{rc}(n, (\mathrm{Dt}, \mathcal{S}', \mathcal{R})) \\
&= 1 + \mathrm{rc}(n, (\mathrm{Dt}, \{s\}, \mathcal{R})) + c_1 + c_2 \cdot \mathrm{rc}^{\mathsf{c}}(n, (\mathrm{Dt}, \mathcal{S}', \mathcal{R})) && (\dagger) \\
&= 1 + \mathrm{rc}(n, (\mathrm{Dt}, \{s\}, \mathcal{R})) + c_1 + c_2 \cdot (\mathrm{rc}^{\mathsf{c}}(n, P) + \mathrm{rc}(n, (\mathrm{Dt}, \{s\}, \mathcal{R}))) && (\ddagger) \\
&\leq 1 + c_1 + (1 + c_2)\mathrm{rc}(n, (\mathrm{Dt}, \{s\}, \mathcal{R})) + c_2 \cdot \mathrm{rc}^{\mathsf{c}}(n, P) \\
&\leq 1 + c_1 + (1 + c_2)(1 + k \cdot \mathrm{rc}^{\mathsf{c}}(n, P)) + c_2 \cdot \mathrm{rc}^{\mathsf{c}}(n, P) \\
&\leq (2 + c_1 + c_2) + (k + kc_2 + c_2) \cdot \mathrm{rc}^{\mathsf{c}}(n, P) \\
&\leq c_1' + c_2' \cdot \mathrm{rc}^{\mathsf{c}}(n, P)
\end{aligned}$$

for some constants c_1' and c_2'. In the (†)-step, we are applying the induction hypothesis, while in the (‡)-step we use the fact that $(\mathrm{Dt} \setminus \mathcal{S}) \cup \{s\} = \mathrm{Dt} \setminus \mathcal{S}'$. □

If we fully remove nodes from the graph, there are often nodes, for which some, but not all right-hand sides have no successor anymore. Simplifying those does not influence the complexity of the CDT problem.

Definition 3.22 (Removal of trailing right-hand sides)**.** Let $P = (\mathrm{Dt}, \mathcal{S}, \mathcal{R})$ be a CDT problem. Let $s \to \overline{t}$ with $\overline{t} = [t_1, \ldots, t_n]$ be a node in the dependency graph G such for the right-hand sides with indexes $I = \{1 \leq i_1, \ldots, i_k \leq n\}$ exists no edge in G. Let $\overline{u} = [t_{j_1}, \ldots, t_{j_{n-k}}]$ be a tuple where $j_1 < \ldots < j_{n-k}$ are the indexes not in I. Then the *removing trailing rhs* processor RemoveTrailRhsP returns

$$\textsc{RemoveTrailRhsP}(P) := (\textsc{Poly}(0), (\mathrm{Dt}', \mathcal{S}', \mathcal{R}))$$

where $\mathrm{Dt}' := \mathrm{Dt}[s \to \overline{t} / s \to \overline{u}]$ and $\mathcal{S}' := \mathrm{Dt}[s \to \overline{t} / s \to \overline{u}]$.

Lemma 3.23. *The processor* RemoveTrailRhsP *is correct.*

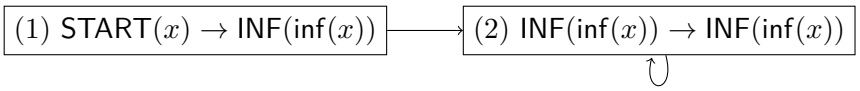
Proof. There is no $(\mathrm{DT}, \mathcal{S}, \mathcal{R})$-t-chain containing $s \to \bar{t}\ \langle i \rangle\ v \to \overline{w}$ for any $i \in I$ and $v \to \overline{w} \in \mathrm{DT}$. □

In particular circumstances, we may also remove nodes without an incoming edge. For removing leading nodes, there are two obstacles: On the one hand, they might have multiple right-hand sides. By removing such a node from a chain tree, we would not only reduce its size by one, but split it into multiple trees. We will see how to handle this case later on. A more serious issue arises from the fact, that we are concerned with the complexity analysis on basic terms only.

Example 3.24. Consider the nonterminating system $\mathcal{R}$:

$$\begin{aligned} \mathsf{start}(x) &\to \mathsf{inf}(\mathsf{inf}(x)) \\ \mathsf{inf}(\mathsf{inf}(x)) &\to \mathsf{inf}(\mathsf{inf}(x)) \end{aligned}$$

with the following dependency graph:



While there are infinite t-chains consisting only of copies of the second node, none of these starts with a term $t^\sharp$ where t is a basic term. So the runtime complexity of the CDT problem $(\{(2)\}, \mathcal{R})$ is always 0.

So we may not remove a leading node if one of the obstacles above exist.

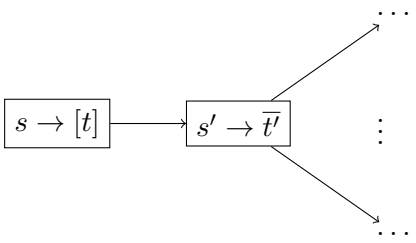
Lemma 3.25 (Removal of simple leading nodes)**.** *Let* $(\mathrm{DT}, \mathcal{S}, \mathcal{R})$ *be a CDT problem with dependency graph* $G = (V, E)$*. Let* $v = s \to [t]$ *be a node of* G *with only one right-hand side. If* v *has no incoming edges and* t *contains no* $\mathcal{D}_\mathcal{R}$*-symbols, then there exist constants* k*,* l *such that*

$$\mathrm{rc}(n, (\mathrm{DT}, \mathcal{S}, \mathcal{R})) \leq 1 + \mathrm{rc}(nk + l, (\mathrm{DT}', \mathcal{S}', \mathcal{R}))$$

holds, where $\mathrm{DT}' := \mathcal{D} \setminus \{v\}$ *and* $\mathcal{S}' := \mathrm{DT}' \cap \mathcal{S}$*.*

Proof. As v has no incoming edge, v can only occur as the root node in a $(\mathrm{DT}, \mathcal{R})$-chain tree. Let T be an innermost $(\mathrm{DT}, \mathcal{R})$-chain tree with substitution σ whose root node is $s \to [t]$ and $s\sigma$ is basic. We now consider the chain tree T' derived from T by removing v. Hence $|T|_\mathcal{S} \leq |T'|_\mathcal{S} + 1$.

If T' is not empty, T has the following form:



There is at least one $(\mathrm{DT}, \mathcal{R})$-chain tree of size 1, namely the tree just consisting of v. Now assume that T has at least size 2. Then, v has exactly one successor in T. As $s\sigma$ is basic and t contains no $\mathcal{D}_\mathcal{R}$ symbols, $t\sigma$ is basic, too and in normal form w.r.t. $\mathcal{R}$. In addition, as $s \to [t], s' \to \overline{t'}$ is a chain, we have $t\sigma = s'\sigma$. So T' is an $(\mathrm{DT}, \mathcal{R})$-chain tree starting with a basic term $(s'\sigma)$, too.

Now, we need to consider how the size of $s\sigma$ and $t\sigma$ are related for arbitrary substitutions σ. Let $x_1, \ldots, x_k$ be the variables of s, occurring $a_1, \ldots, a_k$ in s and $b_1, \ldots, b_k$ times in t. Then $|s\sigma| = |s| = a_1|x_1\sigma| + \ldots + a_k|x_k\sigma|$ and $|t\sigma| = |t| = b_1|x_1\sigma| + \ldots + b_k|x_k\sigma|$. So with $l = \max(0, |s| - |t|)$ and $k = \max\{b_1/a_1, \ldots, b_k/a_k\}$ we have $k|s| + l \geq |t|$. So, if the runtime complexity of $(\mathrm{DT}, \mathcal{R})$ and n is c, there exists a basic term of size at most $kn + l$ such that the size of the maximal chain tree of this tree is at least $c - 1$. Hence, $\mathrm{rc}(n, (\mathrm{DT}, \mathcal{S}, \mathcal{R})) \leq 1 + \mathrm{rc}(nk + l, (\mathrm{DT}', \mathcal{S}', \mathcal{R}))$. □

One should note that for a function f, in general, the asymptotic complexities of $f(nk + l)$ and $f(n)$ are different, in particular we have $\mathcal{O}(f(nk + l)) \not\subseteq \mathcal{O}(f(n))$ in Landau notation. If f is for example the exponential function, we get

$$\frac{2^{nk+l}}{2^n} = \frac{(2^n)^k 2^l}{2^n} = (2^n)^{k-1} 2^l$$

so the factor between $f(nk + l)$ and $f(n)$ is not constant (for $k > 1$). Luckily, for the important case of f being a polynomial function, we get $\mathcal{O}(f(nk + l)) = \mathcal{O}(f(n))$. Therefore we should not apply this technique if we are interested in good bounds other than asymptotic polynomial complexity.

Definition 3.26 (Remove Leading Nodes Processor)**.** Let $(\mathrm{DT}, \mathcal{S}, \mathcal{R})$ be a CDT problem. Let $v = s \to [t]$ be a node of the dependency graph with only one right-hand side. If v has no incoming edges and t contains no $\mathcal{D}_\mathcal{R}$-symbols, then

$$\textsc{RemoveLeadingP}(P) := (\textsc{Poly}(0), \{(\mathrm{DT} \setminus \{v\}, \mathcal{S} \setminus \{v\}, \mathcal{R})\})$$

is the *Remove Leading Nodes* processor.

As $\mathfrak{D}$ contains only polynomial values, the correctness of this processor follows from the previous lemma.

Before, we mentioned handling the problem that a leading node might have multiple right-hand sides. Indeed we can solve a slightly more general problem.

Definition 3.27. Let $s \to \bar{t}$ with $\bar{t} = [t_1, \ldots, t_k]$ be a tuple. Then

$$\mathrm{split}(s \to \bar{t}) := \{s \to [t_1], \ldots, s \to [t_k]\}$$

is the RHS split of $s \to \bar{t}$.

Lemma 3.28 (Split RHSs of nodes)**.** *Let $(\mathrm{DT}, \mathcal{S}, \mathcal{R})$ be a CDT problem with dependency graph G. If $v = s \to [t_1, \ldots, t_k]$, $k > 0$ is a node not part of any SCC, we have*

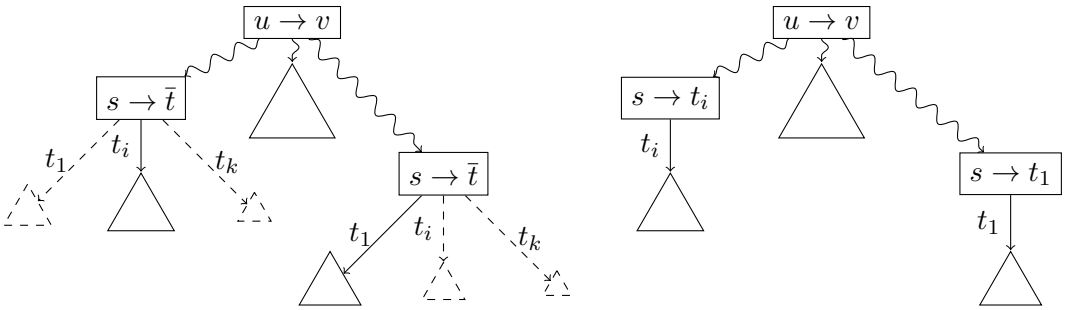
$$\mathrm{rc}(n, (\mathrm{DT}', \mathcal{S}', \mathcal{R})) \leq \mathrm{rc}(n, (\mathrm{DT}, \mathcal{S}, \mathcal{R})) \leq k \cdot \mathrm{rc}(n, (\mathrm{DT}', \mathcal{S}', \mathcal{R}))$$

where $\text{DT}' := (\text{DT} \setminus \{v\}) \cup \mathit{split}(v)$ *and*

$$\mathcal{S}' := \begin{cases} (\mathcal{S} \setminus \{v\}) \cup \mathit{split}(v) & \text{if } v \in \text{DT} \\ \mathcal{S} & \text{if } v \notin \text{DT}. \end{cases}$$

In the case $k = 0$, this split would lead to a complete removal of the node. This case is already handled by Theorem 3.14. As this would make the proof more complicated, we do not include it here.

Proof. Such a node can only occur once in each t-chain. Hence, if we consider all subtrees starting with v in a chain tree, those subtrees are not overlapping. Therefore, if we reduce the size of each subtree by a factor of l, the size of the whole tree is reduced by at most a factor of l. This is demonstrated in the graphs below, where the original chain-tree is on the left and the transformed chain-tree is on the right.



Each subtree T starting with v has at most k successors, each of them again a subtree. Let us call these trees T_1 to T_k. Assume T_l is the largest one (measuring $\mathcal{S}$-nodes) of them. Let T' be the tree we get if we remove all of T_1 to T_k except T_l. Then $|T|_{\mathcal{S}} \leq k|T'|_{\mathcal{S}}$.

Now, if $s \to [t_1, \ldots, t_n], u \to \overline{v}, \ldots$ is a t-chain, then $s \to [t_i], u \to \overline{v}, \ldots$ is also a t-chain for some $1 \leq i \leq n$. So if we replace v by $s \to [t_l]$ in T', the resulting tree is $(\text{DT}', \mathcal{R})$-chain tree. Replacing v does not change the size, as $v \in \mathcal{S}$ if and only if $s \to [t_l] \in \mathcal{S}'$.

So for each $(\text{DT}, \mathcal{R})$-chain tree of size m, we can construct a $(\text{DT}', \mathcal{R})$-chain tree starting with the same term of size m' such that $m < km'$. The two graphs in the figure above visualize this construction. □

The above theorem does not only solve the problem of leading nodes having more then one right-hand side, but leverages also the "defined symbols on the right hand side" problem a bit: If only some of the right-hand sides have defined symbols, we are nevertheless able to delete the others.

Definition 3.29 (RHSSPLITP)**.** Let $P = (\text{DT}, \mathcal{S}, \mathcal{R})$ be a CDT problem. Let V be the set of nodes not part of any SCC and with at least one right-hand side. Let k be the maximal number of right-hand sides of a tuple in $\mathcal{V}$. Then

$$\text{RHSSPLITP}(P) := (f_{\mathsf{up}}, (\text{DT}', \mathcal{S}', \mathcal{R}))$$

where $f_{\mathsf{up}} = \textsc{Poly}(0)$ and

$$\begin{aligned}\textsc{Dt}' &:= \textsc{Dt} \setminus V \cup \{\mathrm{split}(v) \mid v \in V\}\\ \mathcal{S}' &:= \mathcal{S} \setminus V \cup \{\mathrm{split}(v) \mid v \in V \cap \mathcal{S}\}.\end{aligned}$$

Lemma 3.30 (Correctness of RhsSplitP)**.** *The processor* RhsSplitP *is correct.*

Proof. If a node is not part of any SCC, it is still not part of any SCC after splitting the right-hand sides of one node. Hence, we can get to $P' = (\textsc{Dt}', \mathcal{S}', \mathcal{R})$ by multiple application of Lemma 3.28. This yields

$$\mathrm{rc}(n, P) \leq k_1 \cdot \mathrm{rc}(n, P_1) \leq \cdots \leq k_{|V|} \cdot \mathrm{rc}(n, P_{|V|}) = \mathrm{rc}(n, P')$$

for $k_1, \ldots, k_{|V|} \leq k$ and hence $\mathrm{rc}_{\mathfrak{D}}(P) = \mathrm{rc}_{\mathfrak{D}}(P')$. □

Another possibility is to remove nodes which can never occur in a $(\textsc{Dt}, \mathcal{S}, \mathcal{R})$-chain starting with a basic term. Such tuples always have a non-basic left-hand side. If they are not reachable from a node with a basic left-hand side, they can never be used.

Definition 3.31 (RemoveUnusableP)**.** Let $(\textsc{Dt}, \mathcal{S}, \mathcal{R})$ be a CDT problem and V a set of nodes of the dependency graph such that each node has a non-basic left-hand side and all nodes from which V can be reached are in V. Let $f_{\mathsf{up}}(n; r) = r$. Then

$$\textsc{RemoveUnusableP}(P) := (f_{\mathsf{up}}, (\textsc{Dt}', \mathcal{S}', \mathcal{R}))$$

where $\textsc{Dt}' := \textsc{Dt} \setminus V$ and $\mathcal{S}' := \textsc{Dt}' \cap \mathcal{S}$.

The correctness of this processor follows directly from the following lemma.

Lemma 3.32 (Removal of unusable nodes)**.** *Let* $(\textsc{Dt}, \mathcal{S}, \mathcal{R})$ *be a CDT problem and* V *a set of nodes of the dependency graph such that each node has a non-basic left-hand side and all nodes from which* V *can be reached are in* V*. Then*

$$\mathrm{rc}(n, (\textsc{Dt}, \mathcal{S}, \mathcal{R})) = \mathrm{rc}(n, (\textsc{Dt}', \mathcal{S}', \mathcal{R}))$$

where $\textsc{Dt}' := \textsc{Dt} \setminus V$ *and* $\mathcal{S}' := \textsc{Dt}' \cap \mathcal{S}$*.*

Proof. A $(\textsc{Dt}, \mathcal{S}, \mathcal{R})$-t-chain relevant for the runtime complexity of the CDT problem always starts with a basic term and hence with a tuple having a basic left-hand side. As V includes all nodes from which V can be reached (and none of these nodes has a basic left-hand side), there is no $(\textsc{Dt}, \mathcal{S}, \mathcal{R})$-t-chain containing a node in V and starting with a basic term. □

3.3. Transformation techniques

Two nodes are connected in the dependency graph, if there is a chain consisting of those two nodes. So this connection criterion is a local one and only considers a single step. Often, this is less information than what we need to prove a complexity bound (or termination). One can use transformation techniques to look further than one step: A transformation step replaces one node by zero or more new nodes, which incorporate some information about the neighbors of the old node. Often this makes finding a reduction pair easier or even breaks cycles in the dependency graph. The transformations presented here are adapted variants of the Dependency Pair transformations described in Giesl, Thiemann, Schneider-Kamp, and Falke [GTSF06].

Definition 3.33 (Replacing a tuple)**.** Let DT, N be sets of dependency tuples. Let ν be a tuple. Then $\mathrm{DT}[\nu/N]$ is the set, where ν was replaced by N:

$$\mathrm{DT}[\nu/N] := \begin{cases} \mathrm{DT} & \text{if } \nu \notin \mathrm{DT} \\ (\mathrm{DT} \setminus \{\nu\}) \cup N & \text{else} \end{cases}$$

For a CDT problem $(\mathrm{DT}, \mathcal{S}, \mathcal{R})$ we also write $(\mathrm{DT}, \mathcal{S}, \mathcal{R})[\nu/N]$ for $(\mathrm{DT}[\nu/N], \mathcal{S}[\nu/N], \mathcal{R})$.

The idea behind the instantiation techniques is to propagate informations about possible instantiations of variables. For the (backward) instantiation technique one takes preceding nodes $u \to \overline{v}$ into account to instantiate variables of a node $s \to \overline{t}$. If $\overline{v}$ contains no defined symbols, this is easy: v_i and s must unify for some i. But in general, v_i contains $\mathcal{R}$-defined symbols which might get rewritten before s can be applied. So, we need to remove everything from v_i which is not kept constant in a chain $u \to \overline{v}, s \to \overline{t}$. This is done by the Cap function.

Definition 3.34 (Cap)**.** Let $\mathcal{R}$ be a set of generalized rules (i.e., rules potentially violating the variable condition). Then $t' := \mathrm{Cap}_{\mathcal{R}}(t)$ is the term which is derived from t by replacing each subterm $t|_\pi$ by a fresh variable, if there exists a substitution σ such that $t\sigma \to^*_{\mathcal{R}} u \xrightarrow{\pi}_{\mathcal{R}} v$.

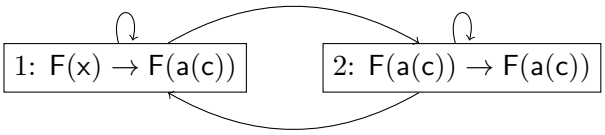
For the innermost case, we have $t' := \mathrm{Cap}^{\mathsf{i}}_{\mathcal{R}}(s \to t)$ as the term, which is derived from t by replacing each subterm $t|_\pi$ by a fresh variable, if there exists a substitution σ such that $s\sigma$ is in $\mathcal{R}$-normal form and $t\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} u \xrightarrow{\mathsf{i}} [\pi]_{\mathcal{R}} v$.

An *estimated cap* function ECap is a function which replaces at least all those positions replaced by the real cap function with fresh variables.

Once again, the definition above is a semantic one. A more general definition (and computable estimations) are given by [GTS05a] and [Thi07], Definition 3.7.

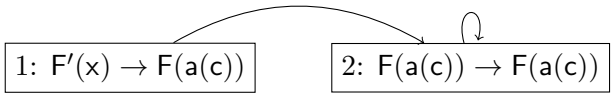
It is important to note that we only analyze rewriting on a restricted set of start terms. So we must make sure that transforming the graph still allows all derivations which are relevant for the complexity of the system, starting with basic terms.

Example 3.35. For illustration, consider the following (incorrect) application of instantiation. Let the CDT problem $(\mathrm{DT}, \mathcal{S}, \mathcal{R})$ be given by the TRS $\mathcal{R} := \{\mathsf{ab} \to \mathsf{b}\}$ and the following dependency graph:



This system has the infinite t-chain $(1), (2), (2), \ldots$ which starts with the basic tuple term $\mathsf{F}(\mathsf{x})$. As $\mathsf{a}(\mathsf{c})$ is in normal form, so we instantiate x with $\mathsf{a}(\mathsf{c})$. Then tuples (1) and (2) are the same and we are left with the system $\{(2)\}$. This system has still an infinite t-chain $(2), (2), \ldots$, but this does not start with a basic term anymore, so it is terminating with regard to the set of basic terms.

To prevent this problem, we always add a new tuple which can be used as a "starter" for such a sequence:



Definition 3.36 (Instantiation Processor)**.** Let $P = (\mathrm{DT}, \mathcal{S}, \mathcal{R})$ be a CDT problem and $s \to \bar{t} \in \mathrm{DT}$ a dependency tuple. Let $\mathrm{ECAP}^{\mathsf{i}}$ be an estimated $\mathrm{CAP}^{\mathsf{i}}$ function and

$$N := \{s' \to \bar{t}\} \cup \{s\mu \to \bar{t}\mu \mid \mu := \mathrm{mgu}(\mathrm{ECAP}^{\mathsf{i}}_{\mathcal{R}}(u \to v_i), s), u \to [v_1, \ldots, v_n] \in \mathrm{DT}, \\ 1 \leq i \leq n, s\mu, u\mu \text{ normal w.r.t. } \mathcal{R}\}$$

where s' is derived from s by replacing the root symbol with a fresh tuple symbol. Then

$$\mathrm{INSTP}(P) := (\mathrm{POLY}(0), (\mathrm{DT}, \mathcal{S}, \mathcal{R})[s \to \bar{t}/N])$$

is the *instantiation processor* INSTP.

Lemma 3.37 (Correctness of INSTP)**.** *The processor* INSTP *is correct.*

Proof. We show that for each $(\mathrm{DT}, \mathcal{S}, \mathcal{R})$-chain tree T there exist a $(\mathrm{DT}', \mathcal{S}', \mathcal{R})$-chain tree T' (and vice versa) such that $|T|_{\{x\}} = |T'|_{\{x\}}$ for all $x \in \mathrm{DT} \cap \mathrm{DT}'$ and that $|T|_{\{s \to \bar{t}\}} = |T'|_N$. Then $\mathrm{rc}_{\mathfrak{D}}((\mathrm{DT}, \mathcal{S}, \mathcal{R})) = \mathrm{rc}_{\mathfrak{D}}((\mathrm{DT}', \mathcal{S}', \mathcal{R}))$ and $\mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}((\mathrm{DT}, \mathcal{S}, \mathcal{R})) = \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}((\mathrm{DT}', \mathcal{S}', \mathcal{R}))$ and the lemma holds true.

Let T' be a $(\mathrm{DT}', \mathcal{R})$-chain tree with substitution σ. The tuple $s' \to \bar{t}$ is a leading node and hence can only occur as the root node of a chain tree. When we replace it by $s \to \bar{t}$ we still have a valid chain tree that still starts with a basic term. Any occurrence of some $s\mu \to \bar{t}\mu$ can be replaced by a variable renamed instance $s'' \to \bar{t}''$ of $s \to \bar{t}$. We just have to extend σ such that $s''\sigma = s\mu\sigma$ and $t''\sigma = t\mu\sigma$.

For the other direction, let T be a $(\mathrm{DT}, \mathcal{R})$-chain tree with substitution σ. We will show that by replacing every occurrence of $s \to \bar{t}$ by a tuple from N we get a $(\mathrm{DT}', \mathcal{R})$-chain tree T' of the same size, starting with a basic term. If the root node is $s \to \bar{t}$, we may replace it by $s' \to \bar{t}$, as shown above. Now consider the other nodes.

Then we have a sub-chain $u \to \overline{v}, s \to \bar{t}$ with $u\sigma$ in normal form and $v_i\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} s\sigma$ for some i. Then $s\sigma$ unifies with $\mathrm{CAP}_{\mathcal{R}}(u \to v_i)$ with an MGU μ. Hence, there is

a substitution θ such that $s\mu\theta = s\sigma$ and $\overline{t}\mu\theta = \overline{t}\sigma$. By renaming $s\mu \to t\mu$ variable disjoint to any tuple in T, the substitution σ can be extended such that $s\mu\theta = s\mu\sigma$ and $\overline{t}\mu\theta = \overline{t}\mu\sigma$. Therefore, we can replace $s \to \overline{t}$ in T' by $s\mu \to \overline{t}\mu$. This does not change the size as $N \in \mathcal{S}'$ if and only if $s \to \overline{t} \in \mathcal{S}$. □

The only difference to the instantiation processor for termination is the additional dependency tuple $s' \to \overline{t}$. This is not necessary for termination analysis, because we are only interested in the existence of a cycle there. For analysis of start term problems this matters as were are interested in the existence of cycles which can be reach from a start term.

The instantiation processor employs information from preceding nodes, so an obvious idea is to do the same thing for succeeding nodes. This complicates matters a bit: Instead of capping the subterms which can be rewritten, we have to cap the terms which might have been the result of rewriting. To do this, we reverse the rules. In general, the resulting rules will not satisfy the variable condition, which is why we defined Cap for generalized rules earlier.

Definition 3.38 (Forward Instantiation Processor)**.** Let $P = (\mathrm{Dt}, \mathcal{S}, \mathcal{R})$ be a CDT problem and $s \to \overline{t} \in \mathrm{Dt}$ a dependency tuple with $\overline{t} = [t_1, \ldots, t_k]$. Let ECap be an estimated Cap function and $\mathcal{EU}$ an estimated usable rules function. Let

$$N := \{s \to []\} \cup \{s\mu \to \overline{t}\mu \mid \mu := \mathrm{mgu}(t_i, \mathrm{ECap}_{\mathcal{R}'^{-1}}(u)), u \to \overline{v} \in \mathrm{Dt}, \\ s\mu, u\mu \text{ normal w.r.t. } \mathcal{R}, 1 \le i \le k\}$$

where $\mathcal{R}' := \mathcal{EU}_{\mathcal{R}}(s \to t)$. Then

$$\mathrm{FInstP}(P) := (\mathrm{Poly}(0), (\mathrm{Dt}, \mathcal{S}, \mathcal{R})[s \to \overline{t}/N])$$

is the *forward instantiation processor* FInstP.

Lemma 3.39 (Correctness of FInstP)**.** *The processor* FInstP *is correct.*

Proof. Like for InstPP, we show that for each $(\mathrm{Dt}, \mathcal{S}, \mathcal{R})$-chain tree T there exist a $(\mathrm{Dt}', \mathcal{S}', \mathcal{R})$-chain tree T' such that $|T|_{\{x\}} = |T'|_{\{x\}}$ for all $x \in \mathrm{Dt} \cap \mathrm{Dt}'$ and that $|T|_{\{s\to\overline{t}\}} = |T'|_N$.

The first case follows with the same reasoning as for Lemma 3.37. For the other direction, let T be an innermost $(\mathrm{Dt}, \mathcal{R})$-chain tree with substitution σ. If there is an occurrence $s' \to \overline{t}'$ of $s \to \overline{t}$ in T and $s\sigma$ is an instance of $s\mu$, we can replace $s \to \overline{t}$ by $s\mu \to \overline{t}\mu$ and still get a valid innermost $(\mathrm{Dt} \cup N, \mathcal{R})$-chain tree: If we extend σ such that $s\mu\sigma = s'\sigma$, then $t\mu\sigma = t\mu$.

Now consider a chain $s \to \overline{t}, u \to \overline{v}$ in T. Then $t\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} u\sigma$ and in particular $t\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}'} u\sigma$. Hence $u\sigma \to^*_{\mathcal{R}'^{-1}} t\sigma$ and therefore $\mathrm{ECap}_{R'^{-1}}(u)$ unifies with t with an MGU μ. As $s\sigma \to \overline{t}\sigma$ and $u\sigma \to \overline{v}\sigma$ are instances of $s\mu \to \overline{t}\mu$ and $u\mu \to \overline{v}\mu$, this means that $s\mu$ and $t\mu$ are in normal form with regard to $\mathcal{R}$ and $s\mu \to \overline{t}\mu \in N$. So we can replace all such inner nodes in T with tuples from N.

All the changes above do not influence the size of the chain tree. The remaining case is if $s \to \overline{t}$ occurs as a leaf node in T. Those can always be replaced by $s \to []$. □

Of course, if we get two MGUs μ_1, μ_2 and μ_1 is more general than μ_2, it suffices to add $s\mu_1 \to \overline{t}\mu_1$ to N, as it is always possible to apply the more general tuple. The same optimization is also possible for the instantiation processor.

Both instantiation techniques work by specializing tuples. Now we present two techniques which work by rewriting the right-hand side of a tuple. The idea of the rewriting processor is fairly simple: If the arguments of a right-hand side are not in normal form, they need to be rewritten before we can reduce the rhs with a tuple, so we want to do the rewrite step in advance. In general, this leads to nondeterminism. But if we know that the usable rules of this right-hand side are non-overlapping, we can always do an arbitrary (not necessarily innermost rewrite step. The adaption of DP-formulation is straight-forward; even the proof does not need many changes.

Definition 3.40 (Rewriting Processor)**.** Let $P = (\textsc{Dt}, \mathcal{S}, \mathcal{R})$ be a CDT problem and $s \to \overline{t} \in \textsc{Dt}$. Let $\overline{t} = [t_1, \ldots, t_n]$. If $U := \mathcal{U}_{\mathcal{R}}(s \to t_i|_p)$ is non-overlapping and weakly innermost terminating for some $1 \leq i \leq k$ and $t_i \xrightarrow{p}_{\mathcal{R}} t_i'$, let $\overline{t}' = [t_1, \ldots, t_{i-1}, t_i', t_{i+1}, \ldots, t_k]\}$. Then the *rewriting processor* is defined as

$$\textsc{RewriteP}(P) := (\textsc{Poly}(0), (\textsc{Dt}, \mathcal{S}, \mathcal{R})[s \to \overline{t}/s \to \overline{t}']).$$

Lemma 3.41 (Correctness of RewriteP)**.** *The processor* RewriteP *is correct*

Proof. We show that $\ldots, s \to \overline{t}\ \langle j \rangle\ u \to \overline{v}, \ldots$ is an innermost t-chain if and only if $\ldots, s \to \overline{t}'\ \langle j \rangle\ u \to \overline{v}, \ldots$ is an innermost t-chain.

As in the proofs for InstP and FInstP, this yields $\mathrm{rc}_{\mathfrak{D}}(P) = \mathrm{rc}_{\mathfrak{D}}((\textsc{Dt}', \mathcal{S}', \mathcal{R}))$ and $\mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P) = \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}((\textsc{Dt}', \mathcal{S}', \mathcal{R}))$.

If $j \neq i$, this is obvious. Otherwise, there exists a substitution σ such that $t_i\sigma = t_i\sigma[t_i|_p\sigma]_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t_i\sigma[r]_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} u\sigma$ with $t_i|_p\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} r$ and r and $u\sigma$ in normal form w.r.t. $\mathcal{R}$.

All rules applicable to $t_i|_p\sigma$ are contained in U. As U is non-overlapping and $t_i|_p\sigma$ is weakly innermost terminating, $t_i|_p\sigma$ is terminating and confluent by [Gra96, Thm 3.2.11]. In particular, r is the unique normal form of $t_i|_p\sigma$. As $t_i|_p \to t_i'|_p$ we have $t_i'|_p \to^*_{\mathcal{R}} r$ and, as r is a normal form, $t_i'|_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} r$. Hence

$$t_i\sigma[t_i'|_p\sigma] \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t_i\sigma[r]_p \xrightarrow{\mathsf{i}}{}^* Ru\sigma$$

and $\ldots, s \to \overline{t}', u \to \overline{v}, \ldots$ is an innermost $(\textsc{Dt}, \mathcal{R})$-t-chain.

For the other direction, the proof is similar: There exists a substitution σ such that $t_i'\sigma = t_i'\sigma[t_i'|_p\sigma]_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t_i'\sigma[r]_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} u\sigma$ with $t_i'|_p\sigma \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} r$ and r and $u\sigma$ in normal form w.r.t. $\mathcal{R}$.

All rules applicable to $t'|_p\sigma$ are contained in U (as $t_i|_p \to_{\mathcal{R}} t'|_p$ holds). Again, as U is non-overlapping and $t_i|_p\sigma$ is weakly innermost terminating, $t_i|_p\sigma$ is terminating and confluent by [Gra96, Thm 3.2.11]. In particular, r is the unique normal form of $t_i|_p\sigma$.

Hence, as $t_i|_p \to_{\mathcal{R}} t'|_p \to^*_{\mathcal{R}} r$, we have $t_i|_p \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} r$, too. Therefore

$$t_i\sigma[t_i|_p\sigma] \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} t_i\sigma[r]_p \xrightarrow{\mathsf{i}}{}^* Ru\sigma$$

and $\ldots, s \to \overline{t}, u \to \overline{v}, \ldots$ is an innermost $(\textsc{Dt}, \mathcal{R})$-t-chain. □

The restriction that $\mathcal{R}$ is weakly innermost terminating can easily be lifted. This requires an asymmetric definition of the complexity, similar to the asymmetric definition of (in)finiteness used in [GTSF06]. It will be easy to adapt this for our framework.

The rewriting technique is related to the narrowing technique. The basic idea is the same: If a right-hand side of a tuple $s \to \bar{t}$ does not unify with any of the successors, at least one $\mathcal{R}$-step has to occur before a DT-step can happen. The REWRITINGP processor handles the case where it is correct to rewrite an arbitrarily chosen redex. But in many cases, the set of usable rules is not non-overlapping. Also, it might be necessary to instantiate some variables before an $\mathcal{R}$-reduction can happen. So for each non-variable position on the right-hand side, the narrowing processor computes all narrowings.

Definition 3.42 (Narrowing of terms and tuples)**.** Let $\mathcal{R}$ be a set of rules, t a term and $\pi \in \mathrm{Pos}(t)$. If $t|_\pi \notin \mathcal{V}$ and t_π unifies with the left-hand side of a (variable-renamed) rule $l \to r \in \mathcal{R}$ and a mgu μ, then $t' := t\mu[r\mu]_\pi$ is a $\mathcal{R}$-*narrowing* of t. If we want to be more exact, we will refer to as a $\pi, l \to r$-narrowing.

If $s \to [t_1, \ldots, t_k] \in \mathrm{DT}$ and t' is an $\mathcal{R}$-narrowing of t_a, $1 \le a \le k$ with substitution μ, then $s\mu \to [t_1\mu, \ldots, t_{a-1}\mu, t', t_{a+1}\mu, \ldots, t_k]$ is a $\mathcal{R}$-*narrowing* of $s \to \bar{t}$.

Recall the formulation of the narrowing processor for Dependency Pairs described in [GTSF06, Def. 28].

Definition 3.43 (Narrowing for Dependency Pairs, [GTSF06, compare Def. 28])**.** Let $\mathcal{P}' = \mathcal{P} \uplus \{s \to t\}$. For $(\mathcal{P}', \mathcal{R})$ the innermost narrowing processor returns $(\mathcal{P} \cup \{s\mu_1 \to t_1, \ldots, s\mu_n \to t_n\}, \mathcal{R})$ if $t_1, \ldots, t_n$ are all $\mathcal{R}$-narrowings of t with the mgu's $\mu_1, \ldots, \mu_n$ such that $s\mu_i$ is in normal form. Moreover, for all $v \to w \in \mathcal{P}'$ where t unifies with the (variable renamed) left-hand side v by a mgu μ, one of the terms $s\mu$ or $v\mu$ must not be in normal form.

This can be adapted with no more than technical changes: We have more then one right-hand side, so we write $\bar{t}$ instead of t. Now, instead of checking whether t unifies with a left-hand side, on has to check whether one of the t_i of $t = [t_1, \ldots, t_n]$ unifies with a left-hand side.

However, having multiple right-hand sides really restricts the applicability: We may not apply narrowing if *one* of the right-hand sides unifies with one of the successor nodes. Also, even if we are allowed to apply narrowing, we have to generate the narrowings for all right-hand sides at once. This is particularly bad, as even narrowing for termination tends to produce many new tuples.

We present an improvement of the narrowing technique which allows narrowing the i-th right-hand side of a tuple $s \to \bar{t}$, if we would be able to narrow the pair $s \to t_i$. The reasoning is the following: Let $\mu_1, \ldots, \mu_m$ be all substitutions of $\mathcal{R}$-narrowings of t_i. If we are allowed to narrow t_i, then t_i does not unify with any variable renamed left-hand side in DT. Now, if we want to rewrite $s\tau$ and $s\tau$ does not match any $s\mu_j$, $1 \le j \le m$, then $t_i\tau$ is normal with regard to $\mathcal{R}$ and hence w.r.t. DT, too. So, we add an additional tuple $s \to \overline{v}$, where $\overline{v}$ is derived from $\bar{t}$ by removing t_i.

Another change is prompted by our proof goal: For narrowing for termination analysis, we only need to care about infinite chains. On the other hand, for complexity analysis

we care in particular aboute finite chains respective chain-trees. So, we need to care about an occurrence of $s \to \bar{t}$ at the end of a t-chain, too: In this case, it is possible that none of the narrowings can replace $s \to \bar{t}$. To make the proof simpler, we add the additional tuple $s \to []$ (which can always replace $s \to \bar{t}$ at the end of a chain, but will never belong to any SCC). In practice, this node can always be removed by applying the SccSplitP processor.

Lemma 3.44 (Narrowing)**.** *Let* $(\mathrm{DT}, \mathcal{S}, \mathcal{R})$ *be a CDT problem and* $s \to \bar{t} \in \mathrm{DT}$ *such that* $\bar{t} = [t_1, \ldots, t_k]$*. Let* $u_1, \ldots, u_m$ *be the narrowings with substitutions* $\mu_1, \ldots, \mu_m$ *of a* t_a*,* $1 \leq a \leq k$ *such that* $s\mu_1, \ldots, s\mu_m$ *are in normal form. Let*

$$\overline{p_j} := [t_1\mu_j, \ldots, t_{a-1}\mu_j, u_j, t_{a+1}\mu_j, \ldots, t_k\mu_j]$$

and let M *be the set of indexes of the right-hand sides which fulfill the following conditions:*

- *For each* $i \in M$*, the term* t_i *does not unify with any left hand side in* DT *and*
- *for each mgu* θ *of the narrowings of* t_i *there is a* j *such that* μ_j *is at least as general as* θ *(i.e., there exists a* σ *such that* $\theta = \mu_j\sigma$*).*

Set $M' := \{1, \ldots, k\} \setminus M$ *and* $\overline{v} := [t_{q_1}, \ldots, t_{q_l}]$ *for* $\{q_1 < \ldots < q_l\} = M'$*.*

If in addition for all $1 \leq i \leq m$ *the right hand side* $t_a\mu_i$ *does not unify with any (variable renamed) left-hand side in* DT*, then*

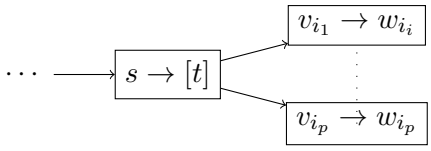
$$\mathrm{rc}(n, (\mathrm{DT}, \mathcal{S}, \mathcal{R})) \leq \mathrm{rc}(n, ((\mathrm{DT}, \mathcal{S}, \mathcal{R})[s \to \bar{t}/N]))$$

for all $n \in \mathbb{N}$*, where*

$$N := \{s \to [], s \to \overline{v}, s\mu_1 \to \overline{u_1}, \ldots, s\mu_m \to \overline{u_m}\}.$$

Proof. We show that for each $t \in \mathcal{T}_B^\sharp$ and for each innermost $(\mathrm{DT}, \mathcal{S}, \mathcal{R})$-chain tree of t there exists a $(\mathrm{DT}', \mathcal{S}', \mathcal{R})$-chain tree of t of at least the same size.

Consider a $(\mathrm{DT}, \mathcal{S}, \mathcal{R})$-chain tree T and an occurrence of $s \to \bar{t}$ in T. If this occurrence has no successor in T, it can be replaced by $s \to []$. Else, it has successors $v_1 \to \overline{w}_1$ to $v_p \to \overline{w}_j$ for the right-hand sides with indexes $1 \leq i_1 < \ldots < i_p < k$ and there is a substitution σ such that $s\sigma$ is in normal form and $t_{i_j}\sigma \xrightarrow{\mathsf{i}}_{\mathcal{R}}^* v_{i_j}\sigma$ for all $1 \leq j \leq p$.



We distinguish three cases:

- $\{i_1, \ldots, i_p\} \cap M = \varnothing$. Then we can replace $s \to \bar{t}$ by $s \to \overline{v}$.

- $\{i_1, \ldots, i_p\} \cap M \neq \varnothing$ and $a \in \{i_1, \ldots, i_p\}$.

  As t_a does not unify with v_a, also $t_a\sigma$ does not unify with $v_a\sigma$. Hence a $l \to r \in \mathcal{R}$-step takes place between them:

  $$t_a\sigma = t_a\sigma[t_a|_\pi\sigma] = t_a\sigma[l\rho]_\pi \xrightarrow{\mathsf{i}} [\pi]_{\mathcal{R}} t_a\sigma[r\rho]_\pi \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} v_a\sigma$$

  We may assume $l \to r$ is renamed variable disjoint to any tuple in the chain tree. Therefore we can extend σ to act on $l \to r$ like ρ, i.e., $l\rho = l\sigma$. Hence there is a substitution τ such that $\mu\tau = \sigma$, where $\mu = \mathrm{mgu}(t_a|_p, l)$.

  Now, $t'_a := t_a\mu[r\mu]_p$ is a narrowing of t. Again, we may assume that $s\mu \to t'_a$ is variable disjoint to any tuple in the chain tree. Therefore we can extend σ to behave like τ on the variables of $s\mu$ (and t'). Then we have $s\mu\sigma = s\sigma$ and

  $$t'_a\sigma = t'_a\tau = t_a\mu\tau[r\mu\tau]_\pi = t_a\sigma[r\sigma]_\pi = t_a\sigma[r\rho]_\pi \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} v_a\sigma$$

  Let $\overline{t'} := [t_1\mu, \ldots, t_{a-1}\mu, t'_a, t_{a+1}\mu, \ldots, t_k]$. Then $s\mu \to \overline{t'}, v_a \to \overline{w_a}$ is still a t-chain. The same holds for $s\mu \to \overline{t'}, v_{i_j}\overline{w_{i_j}}$ for all j with $i_j \neq a$, as $t'_{i_j}\sigma = t_{i_j}\sigma$ holds. If this occurrence of $s \to \bar{t}$ has a predecessor $v' \to \overline{w'}$ in T, this is still a predecessor of $s\mu \to \overline{t'}$, as $s\mu\sigma = s\sigma$. Hence, we may replace $s \to \bar{t}$ by $s\mu \to \overline{t'}$.

- $\{i_1, \ldots, i_p\} \cap M \neq \varnothing$ and $a \notin \{i_1, \ldots, i_p\}$: Let $j \in \{i_1, \ldots, i_p\} \cap M$.

  As t_j does not unify with v_j, also $t_j\sigma$ does not unify with $v_j\sigma$. Hence a $l \to r \in \mathcal{R}$-step takes place between them:

  $$t_j\sigma = t_j\sigma[t_j|_\pi\sigma] = t_j\sigma[l\rho]_\pi \xrightarrow{\mathsf{i}} [\pi]_{\mathcal{R}} t_j\sigma[r\rho]_\pi \xrightarrow{\mathsf{i}}{}^*_{\mathcal{R}} v_j\sigma$$

  We may assume $l \to r$ is renamed variable disjoint to any tuple in the chain tree. Therefore we can extend σ to act on $l \to r$ like ρ, i.e., $l\rho = l\sigma$. Hence there is a substitution τ such that $\theta\tau = \sigma$, where $\theta = \mathrm{mgu}(t_j|_p, l)$.

  There exists a narrowing of t_a with substitution μ such that $\theta = \mu\tau'$ for some τ'. Let $s\mu \to \overline{t'}$ be the tuple belonging to this narrowing. We may assume that $s\mu$ is variable disjoint to any tuple in the chain tree and hence we can extend σ to act like $\tau'\tau$ on $s\mu$. Then we have

  $$s\mu\sigma = s\mu\tau'\tau = s\theta\tau = s\sigma$$

  and we can replace $s \to t$ by $s\mu \to \overline{t'}$.

□

Definition 3.45 (Narrowing Processor)**.** Let $P = (\textsc{Dt}, \mathcal{S}, \mathcal{R})$ be a CDT problem and let $s \to \bar{t}$ and N be defined like in the previous lemma. If $s \to \bar{t}$ and all nodes of the dependency graph reachable from $s \to \bar{t}$ are in $\mathcal{S}$, then

$$\textsc{NarrowingP}(P) := (\textsc{Poly}(0), (\textsc{Dt}[s \to \bar{t}/N], \mathcal{S}, \mathcal{R}))$$

is the narrowing processor NarrowingP.

Lemma 3.46 (Correctness of NarrowingP)**.** *The processor* NarrowingP *is correct.*

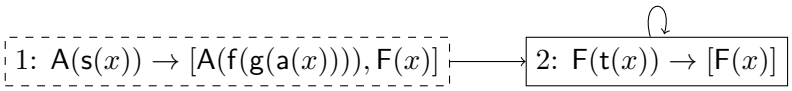
Proof. Follows directly from Lemma 3.44, as $\mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(P) = \mathrm{rc}^{\mathsf{c}}_{\mathfrak{D}}(\mathrm{DT}[s \to \bar{t}/N], \mathcal{S}, \mathcal{R})$. □

Restricting the narrowing processor to cases where all rules reachable from the narrowed tuple are in $\mathcal{S}$ is a pretty severe applicability restriction. In particular, a single successful application of the reduction pair processor on a SCC forbids the use of the narrowing processor on this SCC later on. This is caused by NarrowingP not being *complete* (i.e., the transformation may increase the complexity): If this increases the complexity of a known node, the known node could hide the complexity of an unknown node.

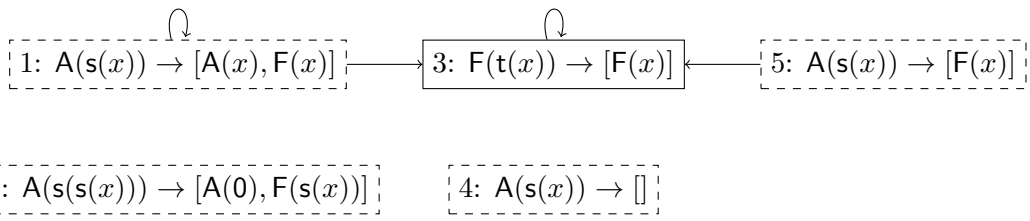
Example 3.47. Consider the CDT problem with rules

$$\mathsf{f}(\mathsf{g}(\mathsf{a}(x))) \to x$$
$$\mathsf{g}(\mathsf{a}(\mathsf{s}(x))) \to \mathsf{0}$$

and the following dependency graph



This CDT problemhas a linear runtime complexity. Note that there is no loop from node (1) to itself, as $\mathsf{f}(\mathsf{g}(\mathsf{a}(x)))$ can never be innermost evaluated to a with root symbol s. If we forgot for a moment about the restrictions related to $\mathcal{S}$, we would be allowed to apply narrowing to node (1), yielding the following new dependency graph:



This new system still has a linear runtime complexity. But now node (1) has linear complexity, too. As it is not in $\mathcal{S}$, a processor working on the new problem is allowed to return Poly(0), which would be incorrect.

Of course, one can always enlarge $\mathcal{S}$ before applying the processor, but this is of dubious value: We have to prove the complexity for those nodes again. So we propose two approaches to make the narrowing processor more applicable: A first one would be to find additional conditions such that this processor is indeed complete. If $\mathcal{R}$ is innermost terminating, the conditions of [GTSF06, Def. 28a] will work; but this would require an additional termination proof. In the cited paper, this is avoided by an asymmetric definition of (in)finiteness for Dependency Pair problems. This approach should be adapted to our framework.

Our second suggestion is more along the lines of "enlarging $\mathcal{S}$": Instead of splitting $\mathrm{D_T}$ in the two sets of known nodes (those in $\mathcal{S}$) and unknown nodes (those in $\mathrm{D_T} \setminus \mathcal{S}$), one could use a ternary split:

- *unknown nodes*: We still need to determine how often those nodes can occur. This is equivalent to our current $\mathcal{S}$.
- *known nodes*: We already know how often those nodes can occur. This is equivalent to our current $\mathrm{D_T} \setminus \mathcal{S}$.
- *ignored nodes*: We have already proved how often these nodes can occur *in the original system*, but we are not allowed to use this information to prove the complexity of other nodes, as it may be different *in the current system*.

Then the narrowing processor would be correct, if it changed the status of all known nodes reachable from the narrowed node to counted.

3.4. A strategy for applying processors

Until now, we have developed a quite a list of processors: In Chapter 2, the processors UsableRulesP an RedPairP were presented. The processors KnownnessP, SccSplitP, RemoveTrailRhsP, RhsSplitP and RemoveUnusableP presented in Section 3.1 and 3.2 always return simpler problems. This is not necessarily the case for the processors InstP, FInstP, RewriteP, NarrowingP presented in Section 3.3. Often they allow solving CDT problems which cannot be proved with only the other techniques. But all of them can be applied infinitely on certain problems without simplifying the problem, but making it harder to solve.

If no start term transformation is performed (compare Chapter A), the first processors should be RemoveUnusableP and RemoveLeadingP, as they possible remove tuples completely from the problem. The RhsSplitP processor potentially increases the size of the graph, but the resulting tuples are clearly simpler to solve, so it should be applied after the RemoveUnusableP processor as long as applicable. The SccSplitP processor only should be used afterwards, otherwise the previous processors would have to be applied to all the new problems. This should be followed by RemoveTrailRhsP as long as applicable, as the split may make additional right-hand sides trailing. As the UsableRulesP processor yields better results for smaller sets of tuples, it is sensible to apply this processor only afterwards.

These initial transformations are guaranteed to make the CDT problem simpler. As searching for orders is costly compared to the other processors, one should apply the RedPairP processor not until now. A successful application of his processor should be always followed by as many KnownnessP passes as possible. It may be useful to restrict the runtime of the reduction pair processor, as this can vary greatly for current search techniques. If the reduction pair processor is not successful, one of the transformation processors should be tried. As those techniques may simplify cycles in the graph, it is advantageous, to use the knownness propagation, RemoveUnusableP and SccSplitP

processors afterwards, followed again by the RedPairP processor. A simple method to prevent infinite applications of the transformation processors is applying them at most a constant number of times. A much more sophisticated heuristic is described in [GTSF06].

4. Annotated CDTs

In Example 2.17 we remarked that the CDT transformation does not exactly model the derivations possible in the original system. The transformed system allows derivations which have no equivalent in the original system. This can even lead to an exponential blowup of the complexity.

Example 4.1. Consider the following TRS, which has a linear runtime complexity.

$$\mathsf{g}(x) \to x \tag{4.1}$$
$$\mathsf{g}(x) \to \mathsf{a}(\mathsf{f}(x)) \tag{4.2}$$
$$\mathsf{f}(\mathsf{s}(x)) \to \mathsf{f}(\mathsf{g}(x)) \tag{4.3}$$

This can be easily seen by the following polynomial interpretation (see also 2.56):

$$[\mathsf{g}](x) = x + 1 \qquad [\mathsf{a}](x) = x \qquad [\mathsf{f}](x) = x \qquad [\mathsf{s}](x) = x + 2$$

But the transformed system with the dependency tuples

$$\mathsf{G}(x) \to [] \tag{4.4}$$
$$\mathsf{G}(x) \to [\mathsf{F}(x)] \tag{4.5}$$
$$\mathsf{F}(\mathsf{s}(x)) \to [\mathsf{F}(\mathsf{g}(x)), \mathsf{G}(x)] \tag{4.6}$$

admits a exponential runtime complexity, as the following derivation shows:

$$\begin{aligned}
\mathsf{F}(\mathsf{s}^{\mathsf{n}+1}(x)) &\xrightarrow{\mathsf{i}} [F(s^n(x)), G(s^n(x))] \\
&\xrightarrow{\mathsf{i}} [F(g(s^n(x))), G(s^n(x))] \\
&\xrightarrow{\mathsf{i}} [F(s^n(x)), G(s^n(x))] \\
&\xrightarrow{\mathsf{i}} [F(s^n(x)), [F(s^n(x))]]
\end{aligned}$$

Only one s-symbol was removed, but the number of F-terms was duplicated.

While the above example is a pathological one (it employs nested recursion [Tai61], which is not often seen in real programs), the asymptotic complexity may increase even for more "normal" examples. Also, the transformation grows the size of the right-hand side considerably, which is a disadvantage when trying to find polynomial interpretations.

Example 4.2. Example 2.11 has a linear complexity which cannot be shown by any combination of CDT techniques and linear polynomial interpretations as described in 2.56. This boils down to the inability to find a suitable interpretation for

$$\mathsf{minus}(x, \mathsf{0}) \to x \tag{4.7}$$
$$\mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{minus}(x, y) \tag{4.8}$$
$$\mathsf{MINUS}(\mathsf{s}(x), \mathsf{s}(y)) \to [\mathsf{MINUS}(x, y)] \tag{4.9}$$
$$\mathsf{QUOT}(\mathsf{s}(x), \mathsf{s}(y)) \to [\mathsf{QUOT}(\mathsf{minus}(x, y), \mathsf{s}(y)), \mathsf{MINUS}(x, y)] \tag{4.10}$$

which orients 4.9 strictly and the other rules weakly. Now, if we use the WiDP approach presented by Hirokawa and Moser [HM08a], we have to replace the tuple 4.10 by

$$\mathsf{QUOT}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{QUOT}(\mathsf{minus}(x, y), \mathsf{s}(y)) \tag{4.11}$$

and orient all rules strictly. This can be done with a linear interpretation.

If we would know that the subterms starting with minus and MINUS in 4.10 have to be evaluated the same way, we would by allowed to (implicitly) transform our CDT problem in a WiDP problem for finding an interpretation. We will call such subterms *related*. Note that without any transformation, the original TRS from Example 2.11 cannot even be shown to be terminating with a polynomial interpretation.

Both problems motivate the introduction of *Annotated CDT problems*. The idea is to keep track which application of a CDT corresponds to which $\mathcal{R}$-rule. For this, we introduce the notions of *correspondence labeled terms* and *lockstepped rewriting*. These changes result in transformation, which exactly models the derivations and runtime complexity of the original system.

4.1. Annotated Complexity Dependency Tuples

To keep track which tuples corresponds to which rule, we will need some additional informations. First, the rule a tuple is based on. Usually that is the rule which was used to generate the ACDT during the transformation process. Second, we need to know which defined positions of the base rule actually occur in the tuple. And third, we need to keep track of substitutions applied to the tuple. This is mainly necessary to avoid adding additional rules in the Instantiation and Forward Instantiation transformation processors.

Definition 4.3 (Annotated Complexity Dependency Tuple)**.** Let $l \to r \in$ be a rule, σ a substitution and $F \subseteq \mathrm{DPos}(r)$. If $\pi_1 <_{\mathrm{lex}} \cdots <_{\mathrm{lex}} \pi_n$ are the elements of F, then

$$d := l\sigma^\sharp \to [r\sigma|^\sharp_{\pi_1}, \ldots, r\sigma|^\sharp_{\pi_n}]$$

is a CDT based on $l \to r$, σ and F and

$$\mathrm{ACDT}(l \to r, \sigma, F) := (d, (l \to r, \sigma, F))$$

is the corresponding *Annotated Complexity Dependency Tuple*. For a rule $l \to r$, the canonical annotated CDT is defined as $\mathrm{ACDT}(l \to r, \mathrm{id}, \mathrm{DPos}(r))$.

We define some selectors for referring to the elements of an annotated CDT.

Definition 4.4. Let $\nu := (p \to \overline{q}, (l \to r, \sigma, F))$ be an annotated CDT. Then $\mathrm{sel}_{\mathrm{T}}(\nu) := d$ is the *tuple rule* of ν and $\mathrm{sel}_{\mathrm{B}}(\nu) := l \to r$ the *base rule*. The *base substitution* is denoted by $\mathrm{sel}_{\mathrm{S}}(\nu)$ and the set of *base positions* by $\mathrm{sel}_{\mathrm{P}}(\nu)$.

Example 4.5. Consider the rule

$$\mathsf{f}(\mathsf{s}(x), y) \to \mathsf{cons}(\mathsf{g}(x), \mathsf{f}(\mathsf{g}(x), y)$$

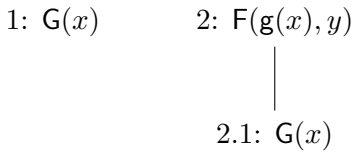
and assume that the function symbols f and g are defined. Then the canonical annotated CDT is

$$\mathsf{F}(\mathsf{s}(x), y) \to [\mathsf{G}(x), \mathsf{F}(\mathsf{g}(x), y), \mathsf{G}(x)]$$

with annotations

$$\mathsf{f}(\mathsf{s}(x), y) \to \mathsf{cons}(\mathsf{g}(x), \mathsf{f}(\mathsf{g}(x), y) \quad \mathrm{id} \quad \{1 <_{\mathrm{lex}} 2 <_{\mathrm{lex}} 2.1\}$$

The set of base positions can be used to reconstruct which right-hand sides are related. The first RHS has position 1, the second 2 and the third 2.1 in the base rule. This relations can also be visualized as a forest (i.e., a set of trees):



We know now that the first right-hand side is independent of the other right-hand sides, but the second and third right-hand side are related (and hence $\mathsf{g}(x)$ should be evaluated the same way as the (second) $\mathsf{G}(x)$.

Furthermore, the tuple $\mathsf{F}(\mathsf{s}(\mathsf{0}), \mathsf{s}(\mathsf{z})) \to [\mathsf{G}(x)]$ with annotations

$$\mathsf{f}(\mathsf{s}(x), y) \to \mathsf{cons}(\mathsf{g}(x), \mathsf{f}(\mathsf{g}(x), y) \quad \{x/\mathsf{0}, y/\mathsf{s}(z)\} \quad \{2.1\}$$

is an example of a non-canonical annotated CDT for the same base rule.

Definition 4.6 (Related positions)**.** Let ν be an ACDTwith tuple rule $p \to [q_1, \ldots, q_n]$. Let $1 \leq i, j \leq n$. Two positions $i.\pi$ and $j.\tau$ are related, if $\kappa_i.\pi = \kappa_j.\tau$ where $\mathrm{sel}_{\mathrm{P}}(\nu) = \{\kappa_1 <_{\mathrm{lex}} \cdots <_{\mathrm{lex}} \kappa_n\}$.

In the example above, the related positions are 2.1 and 3 as well as 2.1.1 and 3.1.

Now we define the annotated variant of CDT problems. For simplicity of the exposition we omit the set $\mathcal{S}$ which was present for ordinary CDT problems. This means that all tuples are always considered to be of unknown complexity. Note that $\mathrm{D_T}$ is now a set of annotated tuples.

Definition 4.7 (Annotated CDT problem)**.** Let $\mathcal{R}$ be a TRS, $\mathrm{D}\mathrm{T}$ be a set of annotated dependency tuples such that $\mathrm{sel}_\mathrm{B}(\nu) \in \mathcal{R}$ for all $\nu \in \mathrm{D}\mathrm{T}$. Then $(\mathrm{D}\mathrm{T}, \mathcal{R})$ is called an annotated CDT problem. It is called *canonical*, if $\mathrm{D}\mathrm{T}$ is the set of canonical ACDTs of $\mathcal{R}$.

For a rule $\rho \in \mathcal{R}$, the set ρ_ass of *associated rules and tuples* contains exactly ρ and all $\nu \in \mathrm{D}\mathrm{T}$ with $\rho = \mathrm{sel}_\mathrm{B}(\nu)$. In addition, we write $\rho \operatorname{ass}_= \rho'$ if either $\rho \in \mathcal{R}$ and $\rho' \in \rho_\mathrm{ass}$ or $\rho = \rho' \in \mathrm{D}\mathrm{T}$.

Note the restriction on $\mathcal{R}$ above. Standard CDT problems do not enforce a certain relationship between $\mathcal{R}$ and $\mathrm{D}\mathrm{T}$. But this restriction is needed, as we often will make use from the fact that a term s is reducible if $s^\sharp$ is reducible. Unfortunately, this prevents using a usable rules processor.

When we rewrite a term with an ACDT, we just handle it like a CDT and ignore the annotations.

Definition 4.8 (Rewriting for annotated CDT problems)**.** Let ν be an ACDT with $\mathrm{sel}_\mathrm{T}(\nu) = p \to \overline{q}$ and let $\rightharpoonup$ be one of $\to$ or $\xrightarrow{\mathsf{i}}$. Then $s \rightharpoonup_\nu t :\Leftrightarrow s \rightharpoonup_{p \to \overline{q}} t$.

So, like with CDTs, rewriting with an annotated CDT will preserve well-formedness of a term. We close this section with a remark about rewriting in an annotated CDT problem. If we can rewrite a sharped term, we always can rewrite the unsharped term to. This is due to the fact that the left-hand sides of tuples are always sharped instances of the left-hand sides of their base rules.

Remark 4.9. Let $(\mathrm{D}\mathrm{T}, \mathcal{R})$ be an annotated CDT problem and $\nu \in \mathrm{D}\mathrm{T}$. If $s^\sharp \xrightarrow{\mathsf{i}}_\nu t$, then also $s \xrightarrow{\mathsf{i}}_{\mathrm{sel}_\mathrm{B}(\nu)} u$ for all terms $s \in \mathcal{T}(\Sigma, \mathcal{V})$. If ν is canonical, even the inverse holds.

4.2. Correspondence Labeling

In Example 4.1 we have demonstrated that the simple transformation of a term rewrite system into a CDT problem may cause an exponential blowup of complexity. We will introduce a method which allows for a complete transformation. The right-hand side of a CDT has independent terms for each defined position. In an ACDT, one can use the base positions to see which one are related.

Now we want to carry over this information to terms which are rewritten with an ACDT. To this end, positions of a term are labeled. The idea is that positions get the same labels if they were created by related positions of an ACDT. This annotation can be used later on to enforce that those subterms are evaluated the same way, thus preventing the exponential complexity blowup.

Definition 4.10 (Correspondence labeling function)**.** A correspondence labeling is a function $\varphi\colon \mathbb{N}^* \to \mathbb{N}^* \cup \{\Box\}$ mapping positions to a label. We call a position π labeled if $\varphi(\pi) \neq \Box$, else *unlabeled*.

The *initial correspondence labeling*, mapping the root position to the empty word and leaving any other position unlabeled, is denoted by ε.

Definition 4.11 (Correspondence labeled terms)**.** A term t with a correspondence labeling φ such that $\varphi(\pi) = \Box$ for all $\pi \notin \mathrm{Pos}(t)$ is called a *correspondence labeled term* or *corlab term*.

We write $t^{\varphi}|_{\pi}$ to denote the corlab term $(t|_{\pi})^{\psi}$ where $\psi(x) := \varphi(\pi x)$.

Example 4.12. Let $t = \textsc{Com}(\mathsf{QUOT}(\mathsf{minus}(x,y),\mathsf{s}(y)),\mathsf{MINUS}(x,y))$ a term with correspondence labeling φ such that $\varphi(1) = 1$, $\varphi(1.1) = \varphi(2) = 2$ and $\varphi(\pi) = \Box$ else. In examples, we write instead

$$t^{\varphi} = \textsc{Com}(\mathsf{QUOT}_1(\mathsf{minus}_2(x,y),\mathsf{s}(y)),\mathsf{MINUS}_2(x,y)).$$

Also, we have $t^{\varphi}|_{1.1} = \mathsf{minus}_2(x,y)$.

If two positions are related in an ACDT, the subterms at this position are almost equal. The only difference is that one of them may be a sharped term.

Definition 4.13 (Term/Tuple Term Equivalence)**.** Let $s,t \in \mathcal{T}(\Sigma,\mathcal{V}) \cup \mathcal{T}^{\sharp}(\Sigma,\mathcal{V})$. We consider s and t *tuple-equivalent*, if $s = t$ or $s \in \mathcal{T}(\Sigma,\mathcal{V})$ and $s^{\sharp} = t$ or $t \in \mathcal{T}(\Sigma,\mathcal{V})$ and $s = t^{\sharp}$. This equivalence relation is denoted by $\equiv_T$.

If s^{φ} and t^{ψ} are correspondence labeled, they are tuple equivalent if and only if $s \equiv_T t$ and $\varphi = \psi$ holds.

At the beginning of this section we explained that same labels shall indicate that those positions were created by related positions of an ACDT. Hence we must define how rewriting a corlab term influences the labels. We will first show how to handle this for rules and then define the labeling for tuples in terms of the labeling of their base rule. For ordinary rules, we just give each defined position an unique label. For tuple rules, the labeling of the subterm corresponding to the position π in the associated rule is to be the same as the labeling of the subterm at π in the associated rule.

Definition 4.14 (Correspondence labeling for rules)**.** Let $l \to r \in \mathcal{R}$ and $w \in \mathbb{N}^*$. Let $\pi_1 <_{\mathrm{lex}}, \cdots <_{\mathrm{lex}} \pi_n$ be the defined positions of r. Then we define $\mathrm{cl}_{\mathcal{R}}(l \to r, w)$ as

$$\mathrm{cl}_{\mathcal{R}}(l \to r, w)(\pi) := \begin{cases} w.i & \text{if } \pi = \pi_i \\ \Box & \text{else.} \end{cases}$$

Example 4.15. Recall the rule from Example 4.5. The positions of a rule are not related, so every position will get a different label. Also, we will only label defined positions, as the others are not relevant for rewriting. To prevent creating duplicates, we prefix the new labels with the old label w at the position of the redex.

$$\mathsf{f}_w(\mathsf{s}(x),y) \to \mathsf{cons}(\mathsf{g}_{w.1}(x),\mathsf{f}_{w.2}(\mathsf{g}_{w.3}(x),y)$$

The labeling of a tuple should match the labeling of the base rule, so that related positions create the same labels. So for the canonical tuple of the above rule, we will define the labeling to be

$$\mathsf{F}_w(\mathsf{s}(x),y) \to [(\mathsf{G}_{w.1}(x),\mathsf{F}_{w.2}(\mathsf{g}_{w.3}(x),y),\mathsf{G}_{w.3}(x)]$$

Please note that $w.i \perp w.j$ for distinct $i, j \in \mathbb{N}$. So, in derivations D_1 and D_2 of e.g. $g_{w.1}$ and $g_{w.3}$, the labels occurring in D_1 will be pairwise distinct from the lables occurring in D_2.

Definition 4.16 (Correspondence labeling for dependency tuples)**.** Let $(\mathrm{DT}, \mathcal{R})$ be an annotated CDT problem, $\nu \in \mathrm{DT}$ a tuple with $\mathrm{sel}_\mathrm{P}(\nu) = \{\kappa_1 <_{\mathrm{lex}} \ldots <_{\mathrm{lex}} \kappa_n\}$ and $\mathrm{sel}_\mathrm{B}(\nu) = \rho$. Let $w \in \mathbb{N}^*$. The correspondence labeling of ν is based on the labeling of ρ:

$$\mathrm{cl}_\mathcal{R}(\nu, w)(\pi) := \begin{cases} \mathrm{cl}_\mathcal{R}(\rho, w)(\pi_j.\tau) & \text{if } \pi = j.\tau \\ \Box & \text{else.} \end{cases}$$

After defining the correspondence labeling of rules, we can now define a labeling rewrite relation. As we are interested in innermost rewriting, we are only interested in the labeling of the root symbol of a redex, so we discard the labeling below. Note that this definition does not restrict rewriting (in comparison to normal innermost rewriting) but just adds a labeling.

Definition 4.17 (Rewrite relation for corlab terms)**.** Let $(\mathrm{DT}, \mathcal{R})$ be a CDT problem and let s, t be terms. For $r \in \mathcal{R} \cup \mathrm{DT}$: $s^\varphi \xrightarrow{\mathrm{i},\pi}_r t^\psi$ iff $s \xrightarrow{\mathrm{i},\pi}_r t$ and

$$\psi(\tau) = \begin{cases} \varphi(\tau) & \text{if } \tau \not\geq \pi \\ \mathrm{cl}(r, \varphi(\pi))(\pi') & \text{if } \tau = \pi\pi' \text{ for some } \pi' \end{cases}$$

Note that all labelings occurring in the original term below the position of the redex are discarded. This is no problem for innermost rewriting, as those terms are known to be in normal form. An appropriate labeling for full rewriting would be slightly more complicated: One would like to preserve the labels, but if the applied rule is duplicating, one would need to use separate labels on each duplicated subterm.

Example 4.18. Let ρ be the rule from 4.15 and assume $\mathsf{g}(\mathsf{0})$ is a normal form. Then

$$\mathsf{h_2}(\mathsf{f_3}(\mathsf{s}(\mathsf{g_4}(\mathsf{0}), y))) \xrightarrow{\mathsf{i}}_\rho \mathsf{h_2}(\mathsf{cons}(\mathsf{g_{3.1}}(\mathsf{g}(\mathsf{0})), \mathsf{f_{3.2}}(\mathsf{g_{3.3}}(\mathsf{g}(\mathsf{0})), y)))$$

is a rewrite step.

If now two positions have the same label, and are rewritten with associated rules, then the labels of the related positions below these positions will be the same: Consider the following rule and its canonical tuple

$$\begin{aligned} \mathsf{g_w}(x) &\to \mathsf{h_{w.1}}(\mathsf{h_{w.2}}(x) \\ \mathsf{G_w}(x) &\to [\mathsf{H_{w.1}}(\mathsf{h_{w.2}}(x), \mathsf{H_{w.2}}(x)]. \end{aligned}$$

Then we have a following derivation:

$$\begin{aligned} [(\mathsf{F_1}(\mathsf{g_2}(x), y), \mathsf{G_2}(x)] &\xrightarrow{\mathsf{i}} [(\mathsf{F_1}(\mathsf{h_{2.1}}(\mathsf{h_{2.2}}(x)), y), \mathsf{G_2}(x)] \\ &\xrightarrow{\mathsf{i}} [(\mathsf{F_1}(\mathsf{h_{2.1}}(\mathsf{h_{2.2}}(x)), y), [\mathsf{H_{2.1}}(\mathsf{h_{2.2}}(x), \mathsf{H_{2.2}}(x)]] \end{aligned}$$

Both the g and G subterms were related in the original term. Rewriting both of them (with associated rules) keeps this property.

In general, we are not interested in all labelings, but only in such labeled terms which could be derived starting from a term t^{ε} or $t^{\sharp\varepsilon}$ for a $t \in \mathcal{T}(\Sigma, \mathcal{V})$. In many cases, we will expect that we rewrite all positions with the same label essentially "at the same time", i.e., if we perform a reduction of a redex labeled w, we will try to reduce all other redexes with the same label before reducing any other position.

Definition 4.19 (Parallel Rewriting)**.** Let $(\mathrm{D}\mathrm{T}, \mathcal{R})$ be an annotated CDT problem and let $s_1 \in \mathcal{T}(\Sigma, \mathcal{V}) \cup \mathcal{T}^{\sharp}(\Sigma, \mathcal{V})$ be a term. Let φ_1 be a correspondence labeling of s. For a $\rho \in \mathcal{R}$ let $s_1^{\varphi_1} \xrightarrow{\mathrm{i},\pi_1}_{\rho_{\mathrm{ass}}} \dots \xrightarrow{\mathrm{i},\pi_n}_{\rho_{\mathrm{ass}}} s_{n+1}^{\varphi_{n+1}}$ be a derivation such that $\varphi_1(\pi_1) = \dots = \varphi_n(\pi_n)$. If all terms labeled with $\varphi_1(\pi_1)$ in $s_{n+1}^{\varphi_{n+1}}$ are in $\xrightarrow{\mathrm{i}}_{\mathrm{D}\mathrm{T}/\mathcal{R}}$-normal form, then it is a *parallel derivation step*. Let $D = D_1 D_2 \dots$ be a derivation such that is D_1 is a parallel derivation step. Then D is called a *parallel derivation*.

The following definition captures the most important properties of such terms.

Definition 4.20 (Correctly corlab terms)**.** Let t^{φ} be a corlab term. We call t^{φ} *correctly corlab* if and only if all of the following restrictions are fulfilled:

(1) Two positions with the same label are equivalent, i.e: Let $\pi, \tau \in \mathrm{Pos}(t)$. If $\varphi(\pi) = \varphi(\tau)$ and $\varphi(\pi) \neq \Box$, then $t^{\varphi}|_{\pi} \equiv_T t^{\varphi}|_{\tau}$.

(2) Labels of tuple symbols are pairwise distinct, i.e: Let $\pi, \tau \in \mathrm{Pos}(t)$ such that $\pi \neq \tau$ and $\mathrm{root}(\pi), \mathrm{root}(\tau) \in \Sigma^{\sharp}$. Then $\varphi(\pi) \neq \varphi(\tau)$.

(3) Smaller labels are in normal form, i.e: Let $\pi, \tau \in \mathrm{Pos}(t)$ be labeled positions. If $\varphi(\pi) \neq \varphi(\tau)$ then either $\varphi(\pi) \perp \varphi(\tau)$ or $\varphi(\pi) < \varphi(\tau)$ and $t^{\varphi}|_{\pi}$ is in normal form with regard to $\mathrm{D}\mathrm{T} \cup \mathcal{R}$ (or vice versa).

(4) All labels below a labeled position are different, i.e: Let $\pi, \tau, \kappa \in \mathrm{Pos}(t)$ be labeled positions with $\pi < \tau, \kappa$ and $\tau \neq \kappa$. Then $\varphi(\tau) \neq \varphi(\kappa)$.

The cases 4.20(1) and 4.20(4) just formalize our intuition that positions labeled the same way should be the same. As with normal CDTs, we only want to count the tuple steps. Hence there should be always only one for each label. For non-canonical systems, a term $t \in \mathcal{T}(\Sigma, \mathcal{V})$ can be reduced if and only iff $t^{\sharp}$ can be reduced, as for each rule $l \to r$ a tuple $l^{\sharp} \to \overline{q}$ exists. For non-canonical systems, this is not guaranteed anymore. So a subterm $t^{\sharp}$ with a label w may remain in the term infinitely, even if other subterms with the same label can be rewritten. This gives rise to case 4.20(3).

We will now see that parallel rewriting preserves the "correctly corlab" property. First a simple remark, which is immediately obvious from the definitions.

Remark 4.21. Let ν be an annotated CDT with $\mathrm{sel}_{\mathrm{T}}(\nu) = p \to q$. Consider the rewrite step $p^{\varphi} \xrightarrow{\pi}_{\nu} q^{\psi}$. Then the labels of tuple symbols in q^{ψ} are pairwise distinct. If ν is a canonical tuple, then each label in ψ occurs also as a label of a tuple symbol in q^{ψ}.

Lemma 4.22. *Let $(\mathrm{D}\mathrm{T}, \mathcal{R})$ be an annotated CDT problem and let $s \in \mathcal{T}(\Sigma, \mathcal{V}) \cup \mathcal{T}^{\sharp}(\Sigma, \mathcal{V})$ be a term. Let $\nu \in \mathcal{R} \cup \mathrm{D}\mathrm{T}$ be a rule. If $s^{\chi} \xrightarrow{\mathrm{i},\varepsilon}_{R} t^{\varphi}$ for some labeling χ, then t^{φ} is correctly corlab.*

We may visualize the proposition of this lemma as: *The rhs of a rule or tuple is correctly corlab.* It is not important whether s^χ is correctly corlab as all labels below the root position are irrelevant.

Proof. Assume that $\nu \in \mathcal{R}$. All labels in $\mathrm{cl}_\mathcal{R}(\nu, \chi(\varepsilon))$ are pairwise distinct, so 4.20(1) holds. By definition of $\mathrm{cl}_\mathcal{R}(\nu, \cdot)$, all labels are pairwise distinct (yields 4.20(2) and 4.20(4)). Furthermore, all labels are independent and therefore 4.20(3) holds.

Otherwise $\nu \in \mathrm{D_T}$ with $\mathrm{sel_B}(\nu) = l \to r$ and $\mathrm{sel_T}(\nu) = l^\sharp\sigma \to [r|^\sharp_{\pi_1}\sigma, \dots, r|^\sharp_{\pi_n}\sigma]$. We have $s = u^\sharp$ for some term u and $u^\chi \xrightarrow{\mathsf{i}}_{l\to r} v^\psi$.

4.20(1) Let $\tau, \kappa \in \mathrm{Pos}(t)$ be distinct positions such that $\varphi(\tau) = \varphi(\kappa)$. Then $\tau = i\tau'$ and $\kappa = j\kappa'$ for some $1 \le i, j \le n$ and $\pi_i\tau' = \pi_j\kappa'$. By construction of ν this yields $t|_\tau \equiv_T t|_\kappa$ and $\varphi|_\tau = \varphi|_\kappa$.

4.20(2) Remark 4.21.

4.20(3) For all labels a, b in t^φ holds either $a = b$ or $a \perp b$.

4.20(4) Let $1 \le i \le n$. Then $t^\varphi|_i \equiv_T v^\psi|_{\pi_i}$ and this property follows as the compound symbol is not labeled. □

The above lemma considers only a single reduction step at the root position. To prove that parallel rewriting in general also preserves this property, we first need a generalization of tuple equivalence: If we rewrite terms s and $s^\sharp$ at the root position, the resulting terms t and u are not tuple equivalent anymore.

Example 4.23. Consider the following rule and its canonical ACDT:

$$\mathsf{q_a}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{s}(\mathsf{q_{a.1}}(\mathsf{m_{a.2}}(x, y), \mathsf{s}(y)))$$
$$\mathsf{Q_a}(\mathsf{s}(x), \mathsf{s}(y)) \to [\mathsf{Q_{a.1}}(\mathsf{m_{a.2}}(x, y), \mathsf{s}(y)), \mathsf{M_{a.2}}(x, y)]$$

If we rewrite the term $\mathsf{q}_\varepsilon(\mathsf{s}(\mathsf{s}(0)), \mathsf{s}(0))$ (and its sharped variant), we get

$$\mathsf{s}(\mathsf{q_1}(\mathsf{m_2}(\mathsf{s}(0), 0), \mathsf{s}(\mathsf{s}(0)))) \text{ respectively } [\mathsf{Q_1}(\mathsf{m_2}(\mathsf{s}(0), 0), \mathsf{s}(0)), \mathsf{M_2}(\mathsf{s}(0), 0)].$$

Now, these terms are not tuple-equivalent anymore, but the subterms with the same labels still are (e.g. $(\mathsf{m_2}(\mathsf{s}(0), 0)$ and $\mathsf{M_2}(\mathsf{s}(0), 0))$. In the above case, both terms still contain the same set of labels. If we use the (non-canonical) tuple

$$\mathsf{Q_a}(\mathsf{s}(x), \mathsf{s}(y)) \to [\mathsf{M_{a.2}}(x, y)]$$

instead, subterms with the same label would still have be tuple-equivalent, but not every label would have a counterpart in the other term.

These cases are formalized in the next definition.

Definition 4.24 (corlab equivalent terms)**.** Let s^φ and t^ψ be terms. Let $(\mathrm{DT}, \mathcal{R})$ be a CDT problem and $s^{\varphi'}$ be derived from s^φ by dropping the labels of subterms in normal form.

If $\operatorname{img}\varphi' \subseteq \operatorname{img}\psi$ and $s|_\pi \equiv_T t|_\tau$ for all positions $\pi \in \operatorname{Pos}(s)$, $\tau \in \operatorname{Pos}(t)$ with $\varphi(\pi) = \psi(\tau) \neq \Box$, then s and t are *(weakly) corlab equivalent* with regard to $(\mathrm{DT}, \mathcal{R})$. This property is denoted by $s^\varphi \sqsubseteq_{\mathrm{cl}} t^\psi$. If even $\operatorname{img}\varphi = \operatorname{img}\psi$, then s^φ and t^ψ are *strongly corlab equivalent*, denoted by $s^\varphi \equiv_{\mathrm{cl}} t^\psi$.

We need to show that the intuition we gave above for the corlab equivalence fits the definition.

Lemma 4.25. *Let $(\mathrm{DT}, \mathcal{R})$ be an annotated CDT problem and let s^φ be a labeled term in $\mathcal{T}(\Sigma, \mathcal{V})$. Let $\rho \in \mathcal{R}$ and let $\nu \in \mathrm{DT}$ be an associated ACDT. If $s^\varphi \xrightarrow{\mathsf{i},\varepsilon}_\rho t^\psi$ and $s^{\sharp\varphi} \xrightarrow{\mathsf{i},\varepsilon}_\nu u^\chi$, then $t^\psi \sqsupseteq_{cl} u^\chi$.*

If ν is canonical, then $s^{\sharp\psi} \xrightarrow{\mathsf{i}}_\nu u^\chi$ is always a derivation and $t^\psi \equiv_{cl} u^\chi$ holds.

Proof. As we are rewriting at ε, we have

$$\psi(\pi) = \mathrm{cl}_\mathcal{R}(\rho, \chi(\varepsilon))(\pi) \text{ for all } \pi \in \operatorname{Pos}(t)$$

respectively

$$\chi(\pi) = \mathrm{cl}_\mathcal{R}(\nu, \chi(\varepsilon))(\pi) \text{ for all } \pi \in \operatorname{Pos}(u).$$

As $\mathrm{cl}_\mathcal{R}(\nu, \cdot)$ is based on $\mathrm{cl}_\mathcal{R}(\rho, \cdot)$, we have $\operatorname{img}\chi \subseteq \operatorname{img}\psi$. Let $\mathrm{sel}_\mathrm{P}(\nu) = \{\tau_1 <_{\mathrm{lex}} \cdots <_{\mathrm{lex}} \tau_n\}$ be the base positions of ν.

We still need to show that $t|_\pi \equiv_T u|_\kappa$ holds for all $\pi \in \operatorname{Pos}(t)$, $\kappa \in \operatorname{Pos}(u)$ with $\psi(\pi) = \chi(\kappa)$. In this case there exist $1 \leq i, j \leq n$ such that $\pi = \tau_i$ and $\kappa = j\kappa'$ and $\tau_j\kappa' = \tau_i$ by the definition of cl_R. Hence

$$u|_\kappa = (u|_j)|_{\kappa'} \equiv_T (t|_{\tau_j})|_{\kappa'} = t|_{\tau_j\kappa'} = t|_{\tau_i} = t|_\pi$$

and $t^\psi \sqsubseteq_{\mathrm{cl}} u^\chi$ is proven.

If ν is canonical then $s^{\sharp\psi} \xrightarrow{\mathsf{i}}_\nu u^\chi$ is a derivation by Remark 4.9. In this case $\tau_1, \ldots, \tau_n$ are all defined positions of the DPos of the right-hand side of ρ and therefore

$$\psi(i) = \mathrm{cl}_\mathcal{R}(T, \chi(\varepsilon))(i) = \mathrm{cl}_\mathcal{R}(R, \chi(\varepsilon))(\tau_i) = \varphi(\tau_i)$$

for all $1 \leq i \leq n$ and we derive $\operatorname{img}\varphi = \operatorname{img}\psi$, hence $t^\varphi \equiv_{\mathrm{cl}} u^\psi$. □

Lemma 4.26 (Rewriting preserves correct correspondence labeling)**.** *Let $(\mathrm{DT}, \mathcal{R})$ be an annotated CDT problem and let s^φ be a correctly corlab term from $\mathcal{T}(\Sigma, \mathcal{V}) \cup \mathcal{T}^\sharp_T(\Sigma, \mathcal{V})$. Choose an $a \in \operatorname{img}\varphi$. Let $\Pi := \{\pi_1, \ldots, \pi_m\}$ be the set of all positions of s such that $\varphi(\pi_i) = a$ and $s|_{\pi_i}$ not in normal form. If $s^\varphi \xrightarrow{\mathsf{i},\pi_1}_{\rho_1} \cdots \xrightarrow{\mathsf{i},\pi_m}_{\rho_m} t^\psi$ and $\rho_i \in \rho_{ass}$ for some $\rho \in \mathcal{R}$, then t^ψ is correctly corlab.*

Proof. First some auxiliary propositions:

† As s^φ is correctly corlab, $s|_\kappa \equiv_T s|_\tau$ for all $\kappa, \tau \in \Pi$. By Lemma 4.25 we derive $t^\psi|_\kappa \sqsubseteq_{\mathrm{cl}} t^\psi|_\tau$ or $t^\psi|_\kappa \sqsubseteq_{\mathrm{cl}} t^\psi|_\tau$.

‡ As any label introduced by $\xrightarrow{\mathsf{i},\pi}$ is strictly greater than $\varphi(\pi)$ and there is no label $u > \varphi(\pi)$ in $\operatorname{img}\varphi$, we have $\psi(\tau) \neq \psi(\kappa)$ for any pair of $\tau < \pi$ and $\kappa \geq \pi$.

We check the conditions of definition 4.20:

4.20(1) Let $\tau, \kappa \in \mathrm{Pos}(t)$ such that $\psi(\tau) = \psi(\kappa)$. For positions "under Π" this condition is obviously satisfied: If $\tau \geq \pi$ and $\kappa \geq \pi'$ for some $\pi, \pi' \in \Pi$ then this property is fulfilled as $t|_\pi \sqsubseteq_{\mathrm{cl}} t|_{\pi'}$ or $t|_\pi \sqsupseteq_{\mathrm{cl}} t|_{\pi'}$ (by †). Else neither $\tau \geq \pi$ nor $\kappa \geq \pi$ for any $\pi \in \Pi$ (because of ‡).

If $\tau, \kappa \perp \pi$ for all $\pi \in \Pi$ then the positions are unaffected by the reduction, i.e., $t^\psi|_\tau = s^\varphi|_\tau$ and $t^\psi|_\kappa = s^\varphi|_\kappa$. Therefore the tuple equivalence property holds, as it held for those positions in s before.

Now assume $\tau < \pi$ and $\kappa < \pi'$ for some $\pi, \pi' \in \Pi$. By definition, these position were equivalent before the rewrite and we have $s|_\tau \equiv_T s|_\kappa$. Hence $s|_\pi = s|_{\pi'}$ follows. By 4.20(2), either $\pi = \pi'$ or $s|_\pi, s|_{\pi'} \notin \Sigma^\sharp$. Therefore $t|_\pi = t|_{\pi'}$. Applying 4.20(4) to s^φ now yields $t|_\tau \equiv_T t|_\kappa$ and $\psi|_\tau = \psi|_\kappa$.

The combination $\tau \perp \pi$ for all $\pi \in \Pi$ and $\kappa < \pi'$ for some $\pi' \in \Pi$ (or the other way round) is not possible. In this case we would have $a \notin \operatorname{img}\varphi|_\tau$ but $a \in \operatorname{img}\varphi|_\kappa$ which is a contradiction to the requirement that s^φ is correctly corlab.

4.20(2) Follows from Lemma 4.21 as there is at most one $1 \leq i \leq m$ such that $\rho_i \in \mathrm{DT}$ because s is correctly corlab.

4.20(3) Let $\Omega := \operatorname{img}(\mathrm{cl}(\rho, a))$. By definition of corlab rewriting, $\operatorname{img}(\psi|_{\pi_i}) \subseteq \Omega$ for all $1 \leq i \leq m$. All labels in Ω are pairwise independent. Furthermore $b > a$ for all $b \in \Omega$. As 4.20(3) holds true in s^φ, we already know that this holds for all labels besides a. But if an a is still in t^φ, it is in normal form.

4.20(4) Subterms starting at positions in Π are correctly corlab by Lemma 4.22. Subterms starting at positions independent of all elements of Π were correctly corlab in s^φ and stay so in t^ψ. Therefore we only need to consider positions above positions in Π.

If a position τ above a tuple symbol labeled a is labeled, there can be no other position with label $\varphi(\tau)$ (as only one tuple symbol labeled a exists in s^φ by 4.20(2), but by 4.20(1)all terms labeled $\varphi(\tau)$ are equivalent). Therefore, consider $\Pi' := \Pi \setminus \{\pi \mid \mathrm{root}(s|_\pi) \in \Sigma^\sharp\}$. We have $s^\varphi|_\tau = s^\varphi|_{\tau'}$ and hence $t^\psi|_\tau = t^\psi|_{\tau'}$ for all $\tau, \tau' \in \Pi'$. So, this property is preserved, too. □

4.3. Lockstepped Rewriting

In the last section we introduced correspondence labelings and a rewriting relation on corlab terms. Furthermore, we have seen that we can keep the labeling in a special form if

we do parallel rewrite steps. While this is certainly a helpful conception, it is not needed that those steps occur in parallel, they may very well be interleaved with other steps. But we must take care that we do not use different, not associated rules for rewriting redexes with the same label. In this section, we formalize this in the *lockstepped derivation* and show that this rewrite relation can be used to model the runtime complexity of a TRS precisely by an annotated CDT problem.

Which rule or tuple is to be used to reduce a label, is captured by a history function.

Definition 4.27 (History function)**.** Let $(\mathrm{D_T}, \mathcal{R})$ be an annotated CDT problem. A *history function* is a partial mapping $\mathbb{N}^* \to \mathcal{R}$. It associates a label with a rule (and therefore its associated dependency tuples) used when rewriting a subterm with this label.

We denote a history function with an uppercase Greek letter, usually Φ or Ψ. Both the empty history function and the undefined history value are denoted by $\triangle$.

This mapping is enforced and extend by lockstepped rewriting: A rewrite step is allowed, if its an innermost corlab step and either honors the history function or is undefined in the history function. In the latter case, the history function will be extended to remember the choice.

Definition 4.28 (Lockstepped Rewriting)**.** Let $P := (\mathrm{D_T}, \mathcal{R})$ be a CDT problem and let s^φ, t^ψ be corlab terms. Furthermore let Φ, Ψ be history functions and $\nu \in \mathrm{D_T} \cup \mathcal{R}$. Then the innermost lockstepped rewrite relation $\overset{\mathsf{i}}{\mapsto}$ is defined as follows:

$$(s^\varphi, \Phi) \overset{\mathsf{i},\pi}{\longmapsto}_{\{r\}} (t^\psi, \Psi)$$

if and only if $s^\varphi \overset{\mathsf{i},\pi}{\longrightarrow}_{\{r\}} t^\psi$ and one of the following conditions is fulfilled:

(1) The redex is labeled, but the history for this step is undefined, i.e., $\varphi(\pi) \in \mathbb{N}^*$ and $\Phi(\varphi(\pi)) = \triangle$. In this case, we define

$$\Psi(x) = \begin{cases} \nu & \text{if } x = \varphi(\pi) \text{ and } \nu \in \mathcal{R} \\ \mathrm{sel_B}(\nu) & \text{if } x = \varphi(\pi) \text{ and } \nu \in \mathrm{D_T} \\ \Phi(x) & \text{if } x \neq \varphi(\pi). \end{cases}$$

(2) The redex is labeled and the reduction honors the history, i.e. $\varphi(\pi) \neq \square$ and $\Phi(\varphi(\pi)) = r'$ and $r \in r'_{\mathrm{ass}}$.

If we are not interested in the position or the rule used, those may be omitted. The same applies if the concrete history does not matter. In this case we just write $s^\varphi \overset{\mathsf{i}}{\mapsto} t^\psi$.

In chapter 2 we used $(\mathrm{D_T}, \mathcal{R})$-chain trees to measure the complexity of a CDT problem $(\mathrm{D_T}, \mathcal{S}, \mathcal{R})$. For annotated CDT problems $(\mathrm{D_T}, \mathcal{R})$, we will simply use the derivation length of rewrite sequences. To count only $\mathrm{D_T}$- and not $\mathcal{R}$-steps, we make use of *relative rewriting* [BD86].

Definition 4.29 (Relative Rewrite Relation)**.** Let $(\mathrm{DT}, \mathcal{R})$ be an annotated CDT problem. Then

$$\overset{\mathsf{i},\pi}{\longmapsto}_{\mathrm{DT}/\mathcal{R}} := \overset{\mathsf{i}}{\mapsto}_{\mathcal{R}}^{*} \circ \overset{\mathsf{i},\pi}{\longmapsto}_{\mathrm{DT}} \circ \overset{\mathsf{i}}{\mapsto}_{\mathcal{R}}^{*}$$

is the *relative rewrite relation* of DT and $\mathcal{R}$. Here, $\circ$ denotes the composition of orders.

In short, a relative $\mathrm{DT}/\mathcal{R}$-rewrite step consists of zero or more $\mathcal{R}$-steps, exactly one DT-step and zero or more $\mathcal{R}$-steps. In particular, there are only finitely many $\mathcal{R}$-steps in a row. The following theorem suggests the use of this rewrite relation for the definition of complexity for an annotated CDT problem.

Theorem 4.30. *Let $\mathcal{R}$ be a terminating rewrite system, $P := (\mathrm{DT}, \mathcal{R})$ its canonical annotated CDT problem and $t \in \mathcal{T}_{A,\mathcal{R}}$. Then* $\mathrm{dl}((t^{\sharp\varepsilon}, \triangle), \overset{\mathsf{i}}{\mapsto}_{\mathrm{DT}/\mathcal{R}}) = \mathrm{dl}(t, \overset{\mathsf{i}}{\to} R)$.

The proof of this theorem is technically involved. To prove that $\mathrm{dl}((t^{\sharp\varepsilon}, \triangle), \overset{\mathsf{i}}{\mapsto}_{\mathrm{DT}/\mathcal{R}})$ is an upper bound for $\mathrm{dl}(t, \overset{\mathsf{i}}{\to} R)$, we show that each $\overset{\mathsf{i}}{\to}_{\mathcal{R}}$-step can be mimicked by a series of (parallel) lockstepped $\mathrm{DT} \cup \mathcal{R}$-steps and such a series always contains exactly one DT-step.

Lemma 4.31. *Let $(\mathrm{DT}, \mathcal{R})$ be an annotated CDT problem. Let u^{φ} be a correctly corlab term from $\mathcal{T}(\Sigma, \mathcal{V})$ such that each label occurs at most once. Let v^{ψ} be a correctly corlab term and $u^{\varphi} \sqsupseteq_{cl} v^{\psi}$. If $a := \varphi(\pi)$ and*

$$u^{\varphi} \xrightarrow{\mathsf{i},\pi}_{\rho} u'^{\varphi'}$$

for some $\rho \in \mathcal{R}$, then

$$v^{\psi} = v_0^{\psi_0} \overset{\mathsf{i},\tau_1}{\longmapsto}_{\rho_{ass}} \cdots \overset{\mathsf{i},\tau_n}{\longmapsto}_{\rho_{ass}} v_n^{\psi_n}$$

where $\tau_1, \ldots, \tau_n$ all positions in v^{ψ} such that $\psi(\tau_i) = a$ and $v^{\psi}|_{\tau_i}$ not in $\overset{\mathsf{i}}{\mapsto}_{\mathrm{DT}\cup\mathcal{R}}$-normal form. Furthermore, $u'^{\varphi'} \sqsupseteq_{cl} v_n^{\psi_n}$ and both are correctly corlab again.

If $(\mathrm{DT}, \mathcal{R})$ is canonical and $u^{\varphi} \equiv_{cl} v^{\psi}$, then also $u'^{\varphi'} \equiv_{cl} v_n^{\psi_n}$

Proof. Correctly corlab follows from lemma 4.26. By Lemma 4.25, $u'^{\varphi'}|_{\pi} \sqsupseteq_{\mathrm{cl}} v_n^{\psi_n}|_{\tau_i}$ for all $1 \leq i \leq n$. Assume for some i there exist positions $\pi' < \pi$ and $\tau_i' < \tau_i$ such that $\varphi(\pi') = \psi(\tau')$. Then we have $u^{\varphi}|_{\pi'} \equiv_T v^{\psi}|_{\tau_i'}$ and therefore $u^{\varphi}|_{\pi} = v^{\psi}|_{\tau_i}$. As $u \in \mathcal{T}(\Sigma, \mathcal{V})$, both $u^{\varphi}|_{\pi}$ and $v^{\psi}|_{\tau_i}$ are rewritten with ρ. Hence, $u'^{\varphi'}|_{\pi} = v_n^{\psi_n}|_{\tau_i}$ and $u'^{\varphi'}|_{\pi'} = v_n^{\psi_n}|_{\tau_i'}$.

If there is a position τ in v^{ψ} with $\psi(\tau) = a$ such that $\tau \neq \tau_i$ for all $1 \leq i \leq n$, then $v|_{\tau} = v_n|_{\tau}$ is in normal form, hence all labels at and below τ will be dropped for checking whether $v_n^{\psi_n} \sqsubseteq_{\mathrm{cl}} u'^{\varphi'}$. By Remark 4.9, this can only happen if $\mathrm{root}(v|_{\tau}) \in \Sigma^{\sharp}$. In this case, all symbols above are compound symbols. Those do not occur in u, hence they cannot be labeled (or must be in normal form).

As all other positions independent of π respectively τ_i, $1 \leq i \leq n$ have not changed in $u'^{\varphi'}$ respectively $v_n^{\psi_n}$, we have $v_n^{\psi_n} \sqsubseteq_{\mathrm{cl}} u'^{\varphi'}$.

Now show that $u'^{\varphi'} \equiv_{\mathrm{cl}} v_n^{\psi_n}$ if $(\mathrm{DT}, \mathcal{R})$ is canonical and $u^{\varphi} \equiv_{\mathrm{cl}} v^{\psi}$. In this case, we might replace all $\sqsubseteq_{\mathrm{cl}}$ and $\sqsupseteq_{\mathrm{cl}}$ above by $\equiv_{\mathrm{cl}}$. □

Now, we will do the induction to tie the steps described in the previous lemma together. As we only count the DT-steps in the lockstepped derivation, we need to make sure that each label occuring in u also occurs at a tuple symbol in v. This ensures that the lockstepped sequence described in the previous lemma is a single $\overset{\mathsf{i}}{\mapsto}_{\mathrm{DT}/\mathcal{R}}$-step.

Lemma 4.32. *Let $(\mathrm{DT}, \mathcal{R})$ be the canonical annotated CDT problem for the terminating rewrite system $\mathcal{R}$. Let u^φ be a correctly corlab term from $\mathcal{T}(\Sigma, \mathcal{V})$, such that each label occurs at most once. Let v^ψ be a correctly corlab term such that $v \in \mathcal{T}_T^\sharp$ and for each label in $\operatorname{img}\psi \setminus \{\square\}$ there is a position π in v such that root$(v|_\pi)$ is a tuple symbol. Furthermore, let Ψ be a history function such that for each $k \in \operatorname{img}\psi$, we have $\Psi(k') = \triangle$ for all k'. If all rewritable positions in the terms u and v are labeled and $u^\varphi \equiv_{cl} v^\psi$, then*

$$\mathrm{dl}((v^\psi, \Psi), \overset{\mathsf{i}}{\mapsto}_{\mathrm{DT}/\mathcal{R}}) \geq \mathrm{dl}(u^\varphi, \overset{\mathsf{i}}{\to}_{\mathcal{R}})$$

holds.

Proof. We use induction on the derivation length of u^φ. If this is 0, we are done. Else, there is an $u' \in \mathcal{T}(\Sigma, \mathcal{V})$ and a position π such that

$$u^\varphi \overset{\mathsf{i},\pi}{\longrightarrow}_\rho u'^{\varphi'}$$

for some $\rho \in \mathcal{R}$.

Let $\tau_1, \ldots, \tau_n$ be the set of v-positions such that $\psi(\tau_i) = \varphi(\pi)$ for all $1 \leq i \leq n$ for all $1 \leq i \leq n$. As $u \equiv_{\mathrm{cl}} v$ holds, we have $u|_\pi \equiv_T v|_{\tau_i}$. By Remark 4.9, each of those positions can be rewritten either by ρ (if root$(v|_{\tau_i}) \in \Sigma$) or some dependency tuple associated with ρ (if root$(v|_{\tau_i}) \in \Sigma^\sharp$. In addition, we have $\Psi(\psi(\tau_i)) = \triangle$, so we get

$$(v^\psi, \Psi) \overset{\mathsf{i},\tau_1}{\longmapsto}_{\rho_{\mathrm{ass}}} (v_1^{\psi_1}, \Psi_1) \overset{\mathsf{i},\tau_2}{\longmapsto}_{\rho_{\mathrm{ass}}} \cdots \overset{\mathsf{i},\tau_n}{\longmapsto}_{\rho_{\mathrm{ass}}} (v'^{\psi'}, \Psi')$$

as all τ_i are independent.

Exactly one τ_i refers to a position with a tuple symbol in v, so exactly one rule from DT is used and $(v^\psi, \Psi) \overset{\mathsf{i}}{\mapsto}_{\mathrm{DT}/\mathcal{R}} (v'^{\psi'}, \Psi')$ is a single derivation step.

We now need to show that $u'^{\varphi'}$ and $v'^{\psi'}$ fulfill the conditions of this lemma, so we can apply the induction hypothesis. Both terms are correctly corlab by Lemma 4.26 and all rewritable positions are labeled. Rules from $\mathcal{R}$ never insert the same label more than once and, so each label still occurs at most once in u'.

In addition, $\Psi(k') = \triangle$ for all labels k' in φ', ψ'. Remark 4.21 ensures that there is a tuple symbol in v' for each label in ψ'. As both rules and CDTs are $\mathcal{T}_T^\sharp$-preserving, $v' \in \mathcal{T}_T^\sharp$. Also, $\varphi(\psi)$ do not occur anymore in $v'^{\psi'}$, as we have rewritten all positions τ_i for $1 \leq i \leq n$.

By Lemma 4.31 $u'^{\varphi'} \equiv_{\mathrm{cl}} v'^{\psi'}$ and both are correctly corlab. □

This is the first half of theorem 4.30. Note that the proof of this lemma gives us en even stronger result: Not only can every $\overset{\mathsf{i}}{\to}_{\mathcal{R}}$-sequence be mimicked by an $\overset{\mathsf{i}}{\mapsto}_{\mathrm{DT}/\mathcal{R}}$-sequence, but also every parallel $\overset{\mathsf{i}}{\mapsto}_{\mathrm{DT}/\mathcal{R}}$-sequence by an $\overset{\mathsf{i}}{\to}_{\mathcal{R}}$-derivation. Unfortunately, this does

not suffice to prove the other half of theorem 4.30, as not every lockstepped derivation is also a parallel derivation. We will now see that we can reorder any lockstepped rewrite sequence so that subterms with the same label are evaluated in parallel. For canonical CDT problem problems this allows us to derive an $\mathcal{R}$ rewrite sequence of the same length, so we can prove the other direction of Theorem 4.30.

Lemma 4.33. *Let* $(\mathrm{DT}, \mathcal{R})$ *be an annotated CDT problem,* $t_0 \in \mathcal{T}(\Sigma \cup \Sigma^\sharp, \mathcal{V})$*. If*

$$(t_0^{\varphi_0}, \Phi_0) \overset{\mathsf{i},\pi_1}{\mapsto}_{R_1} (t_1^{\varphi_1}, \Phi_1) \overset{\mathsf{i},\pi_2}{\mapsto}_{R_2} (t_2^{\varphi_2}, \Phi_2)$$

for some $R_1, R_2 \in \mathcal{R} \cup \mathrm{DT}$ *and* $\pi_1 \perp \pi_2$*, then also*

$$(t_0^{\varphi_0}, \Phi_0) \overset{\mathsf{i},\pi_2}{\mapsto}_{R_2} (t_1'^{\varphi_1'}, \Phi_1') \overset{\mathsf{i},\pi_1}{\mapsto}_{R_1} (t_2^{\varphi_2}, \Phi_2)$$

Proof. As π_1 and π_2 are independent, we have $t_0 \xrightarrow{\mathsf{i},\pi_2} t_1' \xrightarrow{\mathsf{i},\pi_1} t_2$ for regular rewriting.

At first, we need to check whether the history allows such a swap for lockstepped rewriting. As rewriting changes the history only for labels with undefined history,

$$(t_0^{\varphi_0}, \Phi_0) \overset{\mathsf{i},\pi_2}{\mapsto}_{R_2} (t_1'^{\varphi_1'}, \Phi_1')$$

is a valid derivation. To prove that

$$(t_1'^{\varphi_1'}, \Phi_1') \overset{\mathsf{i},\pi_1}{\mapsto}_{R_1} (t_2^{\varphi_2'}, \Phi_2)$$

is also valid, we consider two cases:

(1) $a := \varphi_0(\pi_1) = \varphi_0(\pi_2)$: We either have $\Phi_0(a) = \triangle$ or $\Phi_0(a) = R$ for an $R \operatorname{ass}_= R_1$. In both cases $\Phi_1(a) = R$ and therefore $R \operatorname{ass}_= R_2$. This allows us to choose $\Phi_1'(a) = R$.

(2) $\varphi_0(\pi_1) \neq \varphi_0(\pi_2)$: We have $\Phi_0(\varphi_0(\pi_1)) = \Phi_1(\varphi_1'(\pi_1))$.

Now, it is left to show that $\varphi_2' = \varphi_2$. But this is obviously true, as the labels of a rewritten subterm only depend on the label and the rule used. □

Now we are going to show that each $\mathrm{DT} \cup \mathcal{R}$-derivation (starting with a sharped basic term) can be reordered such that it contains only parallel derivation steps. By copying the proof of lemma 4.32 this suffices to show a lower bound.

Definition 4.34. For a derivation

$$D := s^\varphi = s_0^{\varphi_0} \overset{\mathsf{i},\pi_1}{\mapsto} s_1^{\varphi_1} \overset{\mathsf{i},\pi_2}{\mapsto} \cdots$$

define $w_D^i := \varphi_{i-1}(\varphi_i)$, i.e., w_D^i is the label of the i-th rewrite step.

Lemma 4.35. *Let* $(\mathrm{DT}, \mathcal{R})$ *be an annotated CDT problem. Let* $t \in \mathcal{T}_T^\sharp(\Sigma, \mathcal{V})$*,* t^ψ *correctly corlab and* D *be a* $\overset{\mathsf{i}}{\mapsto}_{\mathcal{R} \cup \mathrm{DT}}$*-rewrite sequence of* t^ψ*. Then there exists another* $\overset{\mathsf{i}}{\mapsto}_{\mathcal{R} \cup \mathrm{DT}}$*-sequence* D' *with at least the same length such that*

$$D' := t^\psi = t_0^{\psi_0} \overset{\mathsf{i},\tau_1}{\mapsto}_{\rho_1} t_1^{\psi_1} \overset{\mathsf{i},\tau_2}{\mapsto}_{\rho_2} \cdots$$

and the following condition holds:

(1) If $w^i_{D'} \neq w^{i+1}_{D'}$, then $w^i_{D'} \neq w^j_{D'}$ for all $j > i$.

(2) $t^{\psi_0}_0 \xrightarrow{\mathrm{i},\tau_1}_{\rho_1} t^{\psi_1}_1$ is the beginning of D, too.

(3) If $w^i_{D'} \neq w^{i+1}_{D'}$, then $t^{\psi_i}_1$ is correctly corlab.

Proof. Let

$$D := s^{\varphi_0}_0 \xrightarrow{\mathrm{i},\pi_1}_{\rho_1} s^{\varphi_1}_1 \xrightarrow{\mathrm{i},\pi_2}_{\rho_2} \cdots$$

and $t^{\psi} = s^{\varphi_0}_0$. Let $a := w^1_D$. At first, we will show the following proposition:

† If $w^i_D = w^1_D$ for some $1 \leq i \leq n$ then $\pi_j \perp \pi_i$ for all $1 \leq j < i$:

Let $w^i_D = w^1_D = a$ and assume that there is a $j < i$ such that $\pi_j \leq \pi_i$. As t^{ψ} is correctly corlab, there is no no position τ with a label $b < a$ in t^{ψ} such that $t^{\psi}|_{\tau}$ is not normal. Let j be minimal such that $j < i$ and $\pi_j < \pi_i$. Then $t^{\psi}|_{\pi_j}$ is not in normal form and hence not $\psi(\pi_j) < a$. As the labels generated by reducing a redex labeled b are strictly greater then b, it follows that $w^i_D \neq a$. This is a contradiction to our assumption.

Now we show that there is no $j < i$ such that $\pi_j > \pi_i$. We know that $s^{\varphi_0}_0$ is correctly corlab. The step $s^{\varphi_0}_0 \xrightarrow{\mathrm{i},\pi_1} s^{\varphi_1}_1$ is an innermost step, hence all arguments of $s^{\varphi_0}_0|_{\pi_1}$ are in normal form. As $w^i_D = w^1_D$, we have $s^{\varphi_0}_0|_{\pi_1} \equiv_T s^{\varphi_0}_0|_{\pi_i}$ and therefore also the arguments of $s^{\varphi_0}_0|_{\pi_i}$ are in normal form. So there cannot be a rewrite step below π_i before a rewrite step at or above π_i.

We prove this lemma by induction on the length of D. The proposition † and Lemma 4.33 allow us to reorder D such that all rewrites of the label a are at the beginning of the sequence. Thus we arrive at the sequence

$$E := t^{\psi_0}_0 \xrightarrow{\mathrm{i},\tau_1}_{\rho_1} t^{\psi_1}_1 \xrightarrow{\mathrm{i},\tau_2}_{\rho_2} \cdots$$

where $t^{\psi_0}_0 = s^{\varphi_0}_0$ and a k exists such that the first k steps are labeled with a: I.e., $\psi_0(\tau_i) = a$ for $1 \leq i \leq k$ and $\psi_0(\tau_j) \neq a$ for all $j > k$. In particular, we have $\Psi_k = \Phi_1$, as $\rho_1, \ldots, \rho_k$ are all associated.

Consider the subsequence of D' starting with $t^{\psi_k}_k$:

$$E' := t^{\psi_k}_k \xrightarrow{\mathrm{i},\tau_{k+1}}_{\rho_{k+1}} t^{\psi_{k+1}}_{k+1} \xrightarrow{\mathrm{i},\tau_{k+2}}_{\rho_{k+2}} \cdots$$

Now assume that there is still a position τ labeled with a left in $t^{\psi_k}_k$. Then there are two cases:

(1) $t^{\psi_k}_k|_{\tau}$ is in normal form. This is allowed for correctly corlab terms.

(2) $t^{\psi_k}_k|_{\tau}$ is not in normal form. But then there can be no reduction step at a position above τ in the rest of the derivation (as it would not be innermost). So changing this position does not prevent any later rewrite steps.

Hence we can insert a rewrite step $t_k^{\psi_k} \xrightarrow{\mathsf{i}}_{\rho'} t'^{\psi'}$ in E'

$$t_k^{\psi_k} \xrightarrow{\mathsf{i},\tau}_{\rho'} t'^{\psi'} \xrightarrow{\mathsf{i},\tau_{k+1}}_{\rho_{k+1}} t_{k+1}[r]_\tau^{\psi_{k+1}} \xrightarrow{\mathsf{i},\tau_{k+2}}_{\rho_{k+2}} t_{k+2}[r]_\tau^{\psi_{k+2}} \xrightarrow{\mathsf{i},\tau_{k+3}}_{\rho_{k+3}} \cdots$$

where r is the term such that $t_k[r]_\tau = t'$. Afterwards, set $k := k + 1$.

Hence, we can assume that $t_k^{\psi_k}$ is correctly corlab by 4.26. Now, E' fulfills the conditions of this lemma, so we can apply it again. By iterating this (infinitely, if necessary), we finally arrive at a sequence D' which has the required form. □

The next lemma is the counterpart to Lemma 4.32. Note that it is slightly more general, in allowing generic CDT problems, not just canonical ones.

Lemma 4.36. *Let* $(\mathrm{DT}, \mathcal{R})$ *be a CDT problem for a terminating rewrite system* $\mathcal{R}$*. Let* u^φ *be a correctly corlab term from* $\mathcal{T}(\Sigma, \mathcal{V})$ *such that each label occurs at most once. Let* v^ψ *be a correctly corlab term such that* $v \in \mathcal{T}_T^\sharp$*. If all rewritable positions in the terms* u *and* v *are labeled and* $u^\varphi \sqsupseteq_{cl} v^\psi$*, then*

$$\mathrm{dl}(v^\psi, \xrightarrow{\mathsf{i}}_{\mathrm{DT}/\mathcal{R}}) \leq \mathrm{dl}(u^\varphi, \xrightarrow{\mathsf{i}}_{\mathcal{R}}).$$

Proof. We use induction on $\mathrm{dl}(v^\psi, \xrightarrow{\mathsf{i}}_{\mathrm{DT}/\mathcal{R}}) + \mathrm{dl}(u^\varphi, \xrightarrow{\mathsf{i}})$. If the derivation length of v^ψ is 0, we are done. Else, choose a $\xrightarrow{\mathsf{i}}$-derivation D of v^ψ with maximal length. By Lemma 4.35, D can be chosen such that

$$D := v_0^\psi \xrightarrow{\mathsf{i},\tau_1}_{\rho_1} \cdots \xrightarrow{\mathsf{i},\tau_k}_{\rho_k} v_k^{\psi_k} \xrightarrow{\mathsf{i},\tau_{k+1}}_{\rho_{k+1}} \cdots \xrightarrow{\mathsf{i},\tau_n}_{\rho_n} v'^{\psi_n}$$

where $v' \in \mathcal{T}_T^\sharp(\Sigma, \mathcal{V})$ and the following restrictions apply: Let $w := w_D^1$. Then there exists a $1 \leq k \leq n$, such that $w_D^i = w$ for all $1 \leq i \leq k$, but $w_D^i \neq w$ for all $k < i \leq n$. This yields $\rho_1, \ldots, \rho_k \in \rho_{\mathrm{ass}}$ for some $\rho \in \mathcal{R}$. In addition $v_k^{\psi_k}|_\tau$ is in $\xrightarrow{\mathsf{i}}$-normal form for all $\tau \in \mathrm{Pos}(v')$ with $\psi_k(\tau) = w$.

As $u^\varphi \sqsupseteq_{\mathrm{cl}} v^\psi$, there is a position $\pi \in \mathrm{Pos}(u)$ such that $\varphi(\pi) = w$. This position is unique, as each label occurs at most once in u^φ. We have $u^\varphi \xrightarrow{\mathsf{i},\pi}_{\mathcal{R}} u'^{\varphi'}$. Rules from $\mathcal{R}$ never insert the same label more than once. As π is the only labeled position in $\mathrm{Pos}(u^\varphi)$ such that $\varphi(\pi) \perp w$ does not hold, no label occurs more then once in u'.

By Lemma 4.31, $v_k^{\psi_k} \sqsubseteq_{\mathrm{cl}} u'^{\varphi'}$ and both are correctly corlab. As v^ψ is correctly corlab, the $\xrightarrow{\mathsf{i}}_{\mathrm{DT}/\mathcal{R}}$-length of the first k steps of D is 0 or 1. The length of $u^\varphi \to u'^{\varphi'}$ is 1. Therefore we can apply the induction hypothesis. □

Now we are able to prove Theorem 4.30.

Proof Theorem 4.30. We show the theorem by proving that $\mathrm{dl}(t^{\sharp\varepsilon}, \triangle, \xrightarrow{\mathsf{i}}_{\mathrm{DT}/\mathcal{R}})$ is an upper bound for $\mathrm{dl}(t, \xrightarrow{\mathsf{i}} R)$ as well as an lower bound.

The upper bound bound can easily be derived from Lemma 4.32: If t is a normal form, there is nothing left to show. Else, we can use the above lemma to show

$$\mathrm{dl}(t^{\sharp\varepsilon}, \triangle, \xrightarrow{\mathsf{i}}_{\mathrm{DT}/\mathcal{R}}) \geq \mathrm{dl}(t^\varepsilon, \xrightarrow{\mathsf{i}} R).$$

Similary, the lower bound bound can easily be derived from Lemma 4.36: If t is a normal form, there is nothing left to show. Else, we can use the above lemma to show

$$\mathrm{dl}(t^{\sharp\varepsilon}, \triangle, \overset{\mathsf{i}}{\mapsto}_{\mathrm{DT}/\mathcal{R}}) \leq \mathrm{dl}(t^{\varepsilon}, \overset{\mathsf{i}}{\to} R).$$

□

Concluding this section, we have seen that an annotated CDT problem with lock-stepped rewriting accurately models the runtime complexity of the underlying TRS. This comes at the cost of having a more complex tuple variant and a strong relation between $\mathcal{R}$ and DT. We finish with the definition of the complexity of an annotated CDT problem.

4.4. Solving annotated CDT problems with polynomial orders

Until now, we have not seen any technique for actually solving annotated CDT problems. One possibility is convert the problem to an ordinary CDT problem, but this simply discards the additional information in the annotations. We have seen that for canonical annotated CDT problems holds $\mathrm{dl}(t, \overset{\mathsf{i}}{\to}_{\mathcal{R}}) = \mathrm{dl}(t^{\sharp}, \overset{\mathsf{i}}{\mapsto}_{\mathrm{DT}/\mathcal{R}})$. A similar result holds for non-canonical annotated CDT problems .

Remark 4.37. Let $(\mathrm{DT}, \mathcal{R})$ be an arbitrary annotated CDT problem. Then we have $\mathrm{dl}(t^{\sharp}, \overset{\mathsf{i}}{\mapsto}_{\mathrm{DT}/\mathcal{R}}) \leq \mathrm{dl}(t, \overset{\mathsf{i}}{\to}_{\mathcal{R}})$ for all terms $t \in \mathcal{T}_{A,\mathcal{R}}(\Sigma, \mathcal{V})$.

Sketch. Let DT' be the canonical ACDT of $\mathcal{R}$. We show $\mathrm{dl}(t^{\sharp}, \overset{\mathsf{i}}{\mapsto}_{\mathrm{DT}/\mathcal{R}}) \leq \mathrm{dl}(t^{\sharp}, \overset{\mathsf{i}}{\mapsto}_{\mathrm{DT}'/\mathcal{R}})$ holds. For each tuple in $\nu \in \mathrm{DT}$, its base rule $l \to r$ is contained in $\mathcal{R}$, so the canonical tuple ν' for this base rule is in DT'. Let $\pi_1, \ldots, \pi_k$ be the defined positions of r. Then

$$\mathrm{sel}_{\mathrm{T}}(\nu') = l^{\sharp} \to [r|^{\sharp}_{\pi_1}, \ldots, r|^{\sharp}_{\pi_k}]$$

and

$$\mathrm{sel}_{\mathrm{T}}(\nu) = l^{\sharp}\sigma \to [r|^{\sharp}_{\pi_{i_1}}\sigma, \ldots, r|^{\sharp}_{\pi_{i_m}}\sigma]$$

where $1 \leq i_1 < \cdots < i_m \leq k$. So the lhs of ν' matches always if ν matches. Rewriting with ν' yields the same, but maybe more, right-hand sides. □

So the $\mathcal{R}$ part of an annotated CDT problem always contains full information about the complexity of the problem. We can now use this for deciding on a per-tuple basis, if we want to count a function symbol by its tuple symbol (and hence, $\mathrm{DT}/\mathcal{R}$-steps) or by itself (i.e., $\mathcal{R}$-steps). This weakens the restrictions an order must fulfill. Recall the Example 4.2. We remarked that

$$\mathsf{QUOT}(\mathsf{s}(x), \mathsf{s}(y)) \to [\mathsf{QUOT}(\mathsf{minus}(x, y), \mathsf{s}(y)), \mathsf{MINUS}(x, y)]$$

can not be oriented strictly in the setting of the example, but

$$\mathsf{QUOT}(\mathsf{s}(x), \mathsf{s}(y)) \to [\mathsf{QUOT}(\mathsf{minus}(x, y)]$$

can, even if we are required to orient the minus-rules strictly, too. By applying Lemma 4.37 we see that this is allowed; but we have to take care that all the strict steps occurring at minus (or below) must be at monotone positions. The following definition captures all the necessary restrictions.

Definition 4.38. Let $(\mathrm{D_T}, \mathcal{R})$ be a CDT problem. For each pair (ν, i) with $\nu \in \mathrm{D_T}$, $1 \leq i \leq n$ and $\pi_1 <_{\text{lex}} \cdots <_{\text{lex}} \pi_n$ the DPos of ν choose whether $C(\nu, i) = T$ or $C(\nu, i) = (j, \tau)$ where $\pi_j \tau = \pi_i$. Let $\mathcal{EU}$ be an estimated usable rules function.

Let $(\succsim, \succ)$ be a weak reduction pair on $\mathcal{T}^\sharp_T(\Sigma, \mathcal{V})$ such that $\mathcal{EU}_\mathcal{R}(\mathrm{sel_T}(\nu)) \subset \succsim$ for all $\nu \in \mathrm{D_T}$ and $\mathrm{D_T} \subset \succ$ and for each tuple $\nu \in \mathrm{D_T}$ with $\mathrm{sel_T}(\nu) = \nu_l \to \nu_r$ holds

(1) if $C(\nu, i) = T$ then $\mathrm{root}(\nu_r)$ strictly monotonic in its i-th argument and

(2) if $C(\nu, i) = (j, \tau)$ then $\mathrm{root}(\nu_r|_\kappa)$ monotonic at its k-th argument for all proper prefixes $\kappa.k$ of $j.\tau$. Furthermore $\mathcal{EU}_\mathcal{R}(\nu_l \to \nu_r|_{j.\tau}) \subseteq \succ$ and for each rule $l \to r \in \mathcal{EU}_\mathcal{R}(\nu_l \to \nu_r|_{j\tau})$ holds: If π is a defined position in r, then r is monotonic at π.

Remark 4.39. If there is a derivation $s \xrightarrow{\mathsf{i}}_\nu t$ with $\mathrm{sel_T}(\nu) = \nu_l \to \nu_r$ and $C(\nu, i) = (j, \tau)$ holds then for all derivations $t \xrightarrow{\mathsf{i}}{}^*_\mathcal{R} u \xrightarrow{\mathsf{i},>j.\tau}_{l\to r} v$ holds $l \to r \in \mathcal{EU}_\mathcal{R}(\nu_l \to \nu_r|_{j\tau})$ by the definition of usable rules.

We say $\Upsilon(t|_{j.\tau})$ holds if each such rule $l \to r \in \succ$ and the following holds: If π is a defined position in r, then r is monotonic at π.

By Lemma 4.35 we know that all lockstepped derivations can be transformed in the form required by this theorem.

Theorem 4.40. *Let* $(\mathrm{D_T}, \mathcal{R})$ *be a CDT problem and* D *be a parallel lockstepped derivation starting with basic term* s *such that* $D = D_1 D_2 \ldots$*, all derivation steps in* D_i *are labeled with* w_i *and* $w_i \neq w_j$ *for all* $i \neq j$*. Let* $(\succsim, \succ)$ *based on an interpretation* Ξ *be a reduction ordering as described in the previous definition, then* $\mathrm{dl}(s, \overset{\mathsf{i}}{\mapsto}_{\mathrm{D_T}/\mathcal{R}}) \leq [s]$.

Proof. As $(\succsim, \succ)$ is a weak reduction ordering, we have $\succsim$ for each step in D. If each D_i with a DT-step contains at least one $\succ$-step then $[s] \geq \mathrm{dl}(s, \overset{\mathsf{i}}{\mapsto}_{\mathrm{D_T}/\mathcal{R}})$.

We use induction over the prefix $D_1 D_2 \ldots D_n$ of D.

Induction hypothesis: Let t^ψ be the first term of D_n. Then t^ψ is correctly corlab and for each $w \in \mathrm{img}\,\psi \setminus \{\Box\}$ holds one of the following properties:

(1) There is a $\pi \in \mathrm{Pos}(t)$ labeled with w such that π is a monotonic position in t and $\Upsilon(t|_\pi)$ holds.

(2) There is a $\pi \in \mathrm{Pos}(t)$ labeled with w such that $\mathrm{root}(t|_\pi) \in \Sigma^\sharp$, π is a monotonic position in t and $t|_\pi$ is not in normal form.

(3) There exists no $\pi \in \mathrm{Pos}(t)$ labeled with w such that $t|_\pi$ is not in normal form and $\mathrm{root}(t_\pi) \in \Sigma^\sharp$.

If this holds for every D_i, then in each D_i with a DT-step holds (1) or (3) for w_i and therefore there is at least one $\succ$-step in such a D_i.

Note that the first term in each D_i is correctly corlab by Lemma 4.26. Also, all terms are in $\mathcal{T}_T^\sharp$, as this property is preserved by rewriting.

Induction start, i.e., $n = 0$. D_1 consists of only one derivation step, namely $s^\varepsilon \overset{\mathsf{i},\varepsilon}{\mapsto}_\nu t^\psi$ for some t^ψ and $\nu \in \mathrm{DT}$. Let ν_r be the right-hand side of ν and k the arity of the compound symbol of ν_r. Consider all labels w in ψ. Note that each w occurs at most once in t^ψ.

If $\psi(i) \neq w$ for all $1 \leq i \leq k$, then case (3) holds. Otherwise, choose $1 \leq i \leq k$ such that $\psi(i) = w$. If $C(\nu, i) = T$, then case (2) holds if $t|_i$ is not in normal form, else case (3) holds. Now, if $C(\nu, i) = (j, \tau)$, then by Definition 4.38(2), the position $j\tau$ is monotonic and $\Upsilon(t|_j\tau)$ holds, so case (1) holds. So, as t^φ is the start of D_2, the induction start holds.

Induction step, i.e., $n \to n + 1$. Let t^ψ be the first term of D_n and u^χ its last term (and hence the first term of D_{n+1}). As t^ψ is correctly corlab, positions with the same label are independent. As all rewrite steps in D_n are labeled with w_n, this allows us to consider each w_n separately. Consider all labels w in t^ψ.

First, if case (3) holds for a label w in t^ψ, it also holds in u^χ (or w does not occur in u^χ anymore), as no subterms labeled w can be created (as t^ψ is correctly corlab).

Now, assume that (3) does not hold for w. For both (1) and (2) we have to distinguish between $w = w_n$ and $w \neq w_n$.

Case $w \neq w_n$. If (2) holds for w, we have a position π such that $\psi(\pi) = w$ and $\mathrm{root}(t|_\pi) \in \Sigma^\sharp$ and $t|_\pi$ not in normal form. Then all symbols above π are compound symbols, so π is still a monotone position in u and therefore either (2) or (3) holds for w in u^χ.

Otherwise, (1) holds for w in t^φ. Let π be the monotonic position labeled w for which $\Upsilon(t|_\pi)$ holds. We may assume that no rewrite above π occurs in D_n: If a rewrite above π occurs, $t|_\pi$ is in normal form. As $\pi \notin \Sigma^\sharp$ and t^ψ is correctly corlab, all other subterms labeled w are also in normal form and therefore (3) already applies for w in t^ψ.

If a rewrite occurs below π, $\Upsilon(t|_\pi)$ still holds.

Case $w = w_n$. As t^φ is correctly corlab, there is at most one tuple symbol labeled with w in t^ψ. If no such position exists (or it is in normal form), case (3) already holds for this w. So we may assume that there is a DT-step in D_n. Hence there is no such tuple symbol labeled w in u^χ anymore, so (3) holds for w in u^χ.

Now we still need to consider the symbols w occurring in u^χ but not in t^ψ. Consider the case "(1) holds for w_n in t^ψ" first. Let π be a position fulfilling the requirements of (1) and labeled with w_n. Then by Remark 4.39 π is still a monotone position in u^χ and $\Upsilon(u|_\pi)$ holds. By the definition of the correspondence labeling, the labels introduced by a DT-rule are subset of those introduced by an ordinary rule; so for all symbols introduced between t^ψ and u^χ, case (1) holds.

Now consider the case that (2) holds for w in t^ψ. Let π be the position of the tuple symbol. Only those labels introduced by the rewrite step at π are relevant, because (3) applies to the other labels introduced in D_n. As π is a monotonic position, the induction

hypothesis follows with the same argumentation as for the induction start. □

At the beginning of this chapter, we mentioned two TRS with linear complexity which could not be shown by using the CDT techniques presented here. We will now see that they are solvable with our new order for annotated CDT problems.

Example 4.41. Consider the TRS from Example 4.41, which has a linear runtime complexity. The corresponding annotated CDT problem has the following annotated tuples:

$$\begin{aligned} \mathsf{G}(x) &\to \textsc{Com}_1 & \mathsf{g}(x) &\to x, \mathrm{id}, \varnothing \\ \mathsf{G}(x) &\to \textsc{Com}_2(\mathsf{F}(x)) & \mathsf{g}(x) &\to \mathsf{a}(\mathsf{f}(x)), \mathrm{id}, \{1\} \\ \mathsf{F}(\mathsf{s}(x)) &\to \textsc{Com}_3(\mathsf{F}(\mathsf{g}(x)), \mathsf{G}(x)) & \mathsf{f}(\mathsf{s}(x)) &\to \mathsf{f}(\mathsf{g}(x)), \mathrm{id}, \{\varepsilon, 1\} \end{aligned}$$

We use the Com-syntax here, as we need to interpret the compound symbols. The following interpretation makes $\textsc{Com}_3$ not strongly monotonic in the second argument, hence all usable rules of $\mathsf{g}(\mathsf{x})$ must be oriented strictly.

$$\begin{aligned} [\mathsf{F}](x) &= x & [\mathsf{G}](x) &= 1 + x & [\mathsf{a}](x) &= x & [\textsc{Com}_1] &= 0 \\ [\mathsf{f}](x) &= x & [\mathsf{g}](x) &= 1 + x & [\mathsf{s}](x) &= 2 + x & [\textsc{Com}_2](x) &= x \\ & & & & & & [\textsc{Com}_3](x, y) &= x \end{aligned}$$

It is easy to verify that this interpretation fulfills the conditions of Theorem 4.40 and hence linear complexity can be proved for this example.

Example 4.42. Our other introductory example was 2.11. This is linear, even if transformed into a CDT problem. But none of our CDT techniques is able to prove this. Consider the transformation into an annotated CDT problem:

$$\begin{aligned} \mathsf{M}(\mathsf{s}(x), \mathsf{s}(y)) &\to \textsc{Com}_1(\mathsf{M}(x, y)) \\ \mathsf{m}(\mathsf{s}(x), \mathsf{s}(y)) &\to \mathsf{m}(x, y), \mathrm{id}, \{\varepsilon\} \\ \mathsf{Q}(\mathsf{s}(x), \mathsf{s}(y)) &\to \textsc{Com}_2(\mathsf{Q}(\mathsf{m}(x, y), \mathsf{s}(y)), \mathsf{M}(x, y)) \\ \mathsf{q}(\mathsf{s}(x), \mathsf{s}(y)) &\to \mathsf{q}(\mathsf{m}(x, y), \mathsf{s}(y)), \mathrm{id}, \{\varepsilon, 1\} \end{aligned}$$

We removed tuples with an empty right-hand side as they are trailing (which is valid not only for CDT problems but also for the annotated variant considered here). The usable rules are

$$\begin{aligned} \mathsf{minus}(x, 0) &\to x \\ \mathsf{minus}(\mathsf{s}(x), \mathsf{s}(y)) &\to \mathsf{minus}(x, y) \end{aligned}$$

end we find the following linear interpretation fulfilling the conditions of the theorem:

$$\begin{aligned} [M](x, y) &= x & [Q](x, y) &= x & [\textsc{Com}_1](x) &= x \\ [m](x, y) &= 1 + x & [s](x) &= 2 + x & [\textsc{Com}_2](x, y) &= x \end{aligned}$$

Similar to the example above $\textsc{Com}_2$ is not monotonous in its second argument, therefore the usable minus rules have been oriented strictly and occur only at monotonous positions.

4.5. Narrowing Transformation

Like for normal CDT problems, we can also apply transformation techniques to annotated CDT problems. For space reasons, we will only present narrowing here, to illustrate the additional complexity associated with lockstepping. Both instantiation and forward instantiation can be defined the same way as for the unannotated case. This is unproblematic, as substitution automatically takes care of lockstepping (the variables are the same in all right-hand sides). Also, the base rule does not change. But for narrowing and rewriting, this is very involved, if we want to retain the annotations: Instead of just doing one reduction step per narrowing, we need to reduce all related positions of the tuple and the base rule. In most cases, one of the related positions is at a tuple symbol, so we need not only to reduce with rules from $\mathcal{R}$, but also with tuples from DT. But as the structure of our tuple rules is fixed to be one compound symbol around some sharped terms, the right-hand side resulting from the reduction steps needs to be adjusted.

Example 4.43. We will illustrate our techniques with the following example: Let $\mathcal{R}$ be the set

$$\mathsf{g}(x,y) \to \mathsf{f}(\mathsf{times}(x,\mathsf{double}(x))) \tag{4.12}$$
$$\mathsf{double}(\mathsf{0}) \to \mathsf{0} \tag{4.13}$$
$$\mathsf{double}(\mathsf{s}(x)) \to \mathsf{s}(\mathsf{s}(\mathsf{double}(x))) \tag{4.14}$$
$$\mathsf{times}(\mathsf{0},y) \to \mathsf{0} \tag{4.15}$$
$$\mathsf{times}(\mathsf{s}(x),y) \to \mathsf{plus}(y,\mathsf{times}(x,y)) \tag{4.16}$$
$$\mathsf{plus}(x,y) \to \cdots \tag{4.17}$$

and DT the set of the following canonical tuples:

$$\mathsf{G}(x,y) \to [\mathsf{F}(\mathsf{times}(x,\mathsf{double}(y))),\mathsf{TIMES}(x,\mathsf{double}(y)),\mathsf{DOUBLE}(y)] \tag{4.18}$$
$$\mathsf{TIMES}(\mathsf{0},y) \to [] \tag{4.19}$$
$$\mathsf{TIMES}(\mathsf{s}(x),y) \to [\mathsf{PLUS}(y,\mathsf{times}(x,y)),\mathsf{TIMES}(x,y)] \tag{4.20}$$

For space reasons, we will shorten the names of the function symbols to their first character throughout this chapter.

Example 4.44. The first problem we will solve is the structure of a narrowed right-hand side. Consider the rhs of the tuple (4.18):

$$[\mathsf{F}_1(\mathsf{t}_2(x,\mathsf{d}_3(y))),\mathsf{T}_2(x,\mathsf{d}_3(y)),\mathsf{D}_3(y)]$$

Now, if apply the substitution $\{x/\mathsf{s}(z)\}$, we can narrow with (4.16) and its associated tuple:

$$[\mathsf{F}_1(\mathsf{p}_{2.1}(\mathsf{d}(y),\mathsf{t}_{2.2}(z,\mathsf{d}(y)))),\ \ [\mathsf{P}_{2.1}(\mathsf{d}(y),\mathsf{t}_{2.2}(z,\mathsf{d}(y))),\mathsf{T}_{2.2}(z,\mathsf{d}(y))],\ \ \mathsf{D}_3(y)]$$

This cannot be a right-hand side of a tuple, as it is a *nested* list of terms. This can easily solved by *collapsing* the inner list:

$$[\mathsf{F}_1(\mathsf{p}_{2.1}(\mathsf{d}(y),\mathsf{t}_{2.2}(z,\mathsf{d}(y)))),\ \ \mathsf{P}_{2.1}(\mathsf{d}(y),\mathsf{t}_{2.2}(z,\mathsf{d}(y))),\mathsf{T}_{2.2}(z,\mathsf{d}(y)),\ \ \mathsf{D}_3(y)]$$

This operation changes the arity of the outer compound symbol. But this does not affect possible derivations, as compound symbols are only static context.

Definition 4.45. Let s be a term. If there exists a position p and a number i, such that $\mathrm{root}(s|_q)$ a compound symbol for all $q \leq pi$, then the p, i-*collapsing* of s^φ is defined as

$$\begin{aligned}\textsc{Collapse}_{p,i}(s^\varphi) := s^\varphi[\textsc{Com}(&s^\varphi|_{p.1}, \ldots, s^\varphi|_{p.(i-1)},\\ &s^\varphi|_{p.i.1}, \ldots, s^\varphi|_{p.i.m},\\ &s^\varphi|_{p.(i+1)}, \ldots, s^\varphi|_{p.n})]_p\end{aligned}$$

if n is the arity of $\mathrm{root}(s|_p)$ and m the arity of $\mathrm{root}(s|_{p.i})$. We use the Com notation instead of [] here, as it clashes with the replace-in-term notation. We write

$$\textsc{Collapse}_p(s^\varphi) := \textsc{Collapse}_{p,i_1} \circ \cdots \circ \textsc{Collapse}_{p,i_k}$$

where the i_j are all arguments of $s|_p$ such that $\mathrm{root}(s|_{p.i_j})$ is a compound symbol. The symbol $\circ$ denotes function composition.

Remark 4.46. Let $(\mathcal{R}, \mathrm{DT})$ be a CDT problem. In the situation above, if $(s^\varphi, \Phi) \overset{\mathsf{i}}{\mapsto}_{\mathrm{DT}\cup\mathcal{R}}^{*} (t^\psi, \Psi)$ is a derivation, then

$$(\textsc{Collapse}_{p.i}(s^\varphi), \Phi) \overset{\mathsf{i}}{\mapsto}_{\mathrm{DT}\cup\mathcal{R}}^{*} (\textsc{Collapse}_{p.i}(t^\psi), \Psi)$$

is also a derivation.

Example 4.47. Collapsing does not suffice to create a valid rhs. Consider another narrowing of the rhs of tuple (4.18). Using the substitution $\{x/\mathsf{0}\}$ and rule (4.15), we get

$$[\mathsf{F_1}(\mathsf{0}), [], \mathsf{D_3}(\mathsf{0})] \text{ and after collapsing } [\mathsf{F_1}(\mathsf{0}), \mathsf{D_3}(\mathsf{0})]$$

Remember that we also need to narrow the base rule. This gives us $\mathsf{f_1}(\mathsf{0})$. So, no subterm corresponding to $\mathsf{D}(\mathsf{0})$ does occur here, and thus we need to *filter* $\mathsf{D}(\mathsf{0})$ from the narrowed rhs. This leaves us (after collapsing) with $[F_1(0)]$.

The same issue applies to the narrowing from Example 4.44. Here we have $\mathsf{d}(\mathsf{0})$ terms in the narrowed base rule, but they do not correspond to $3D(0)$, as they occur under a rewritten position. So we need to filter it.

Definition 4.48. Let s^φ be a term from $\mathcal{T}_T^\sharp(\Sigma, \mathcal{V})$, $p \in \mathrm{Pos}(s)$ and $s|_p = \textsc{Com}(t_1, \ldots, t_n)$. Let $Q \subseteq \{1, \ldots, n\}$. Then we define

$$\mathrm{filt}_{p,Q}(s^\varphi) := s[\textsc{Com}(t_{i_1}, \ldots, t_{i_m})]_p$$

where $1 \leq i_1 < \cdots < i_m \leq n$ and $\{i_1, \ldots, i_m\} = \{1, \ldots, n\} \setminus Q$.

Example 4.49. So, Collapse and filt can be used to bring a narrowed right-hand side in correct shape. The last issue is labeling. Consider the narrowing from Example 4.44 again. As explained above $\mathsf{D_3}(y)$ needs to be filtered and we have

$$[\mathsf{F_1}(\mathsf{p_{2.1}}(\mathsf{d}(y), \mathsf{t_{2.2}}(z, \mathsf{d}(y)))), \mathsf{P_{2.1}}(\mathsf{d}(y), \mathsf{t_{2.2}}(z, \mathsf{d}(y))), \mathsf{T_{2.2}}(z, \mathsf{d}(y))].$$

This labeling differs from the one which is used when we apply the narrowed rule:

$$\mathsf{G}(\mathsf{s}(z), y) \to [\mathsf{F}_1(\mathsf{p}_2({}_3\mathsf{d}(y), \mathsf{t}_4(z, \mathsf{d}_5(y)))), \mathsf{P}_2(\mathsf{d}_3(y), \mathsf{t}_4(z, \mathsf{d}_5(y))), \mathsf{T}_4(z, \mathsf{d}_5(y)),]$$

But this will not cause a problem: All tuples with the same base rule share the same labeling (and only such with the same base rule can be applied to one specific label).

Having solved these introductory problems, we can now define what a narrowing of an annotated CDT is. For the narrowing of a rule, see Definition 3.42. Narrowing a tuple is more involved. To recapitulate, we have to do the following steps:

(1) Apply the same rule (or an associated tuple) to all related positions.

(2) Filter out those tuples which occur below a narrowed position.

(3) Collapse the nested compound symbols.

The easiest way to do this is building the ACDT from scratch from the narrowed base rule.

Definition 4.50 (Narrowing of an annotated CDT)**.** Let $(\mathrm{D_T}, \mathcal{R})$ be an annotated CDT problem and let $\rho, \nu \in \mathrm{D_T}$ be tuples such that $\mathrm{sel_B}(\nu) = s \to t$. Furthermore, let $l \to r := \mathrm{sel_S}(\rho)(\mathrm{sel_B}(\rho))$ and t' be the $\pi, l \to r$-narrowing with substitution μ of $t\sigma$.

Then the π, ρ-narrowing ν_{nar} of ν is defined as follows:

$$\nu_{\mathsf{nar}} := \mathrm{ACDT}(s\sigma\mu \to t', \mathrm{id}, F)$$

where

$$F := (\mathrm{sel_P}(\nu) \setminus \{\tau \in \mathrm{sel_P}(\nu) \mid \pi < \tau\}) \cup \{\pi\tau \mid \tau \in \mathrm{sel_P}(\rho)\}$$

.

Remark 4.51. Let $(\mathcal{R}, \mathrm{D_T})$ be a CDT problem. Let δ_{nar} be a π, ρ-narrowing of $\delta \in \mathcal{R}$ with $\rho \in \mathcal{R}$ and $s^\varphi \xrightarrow{\mathsf{i},\varepsilon}_\delta u^\chi \xrightarrow{\mathsf{i},\pi}_\rho t^\psi$ a derivation. Then $s^\varphi \xrightarrow{\mathsf{i},\varepsilon}_{\delta_{\mathsf{nar}}} t^{\psi'}$ is also a derivation.

For the next remark we require that all related positions can be rewritten with some ρ_{ass}. We will see later that this does not impose a restriction on the cases we are interested in.

Remark 4.52. Let $(\mathcal{R}, \mathrm{D_T})$ be CDT problem. Let $\nu \in \mathrm{D_T}$. Let

$$P_\pi = \{ip \mid i \in \mathbb{N}, p \in \mathbb{N}^*, \pi_i p = \pi\}.$$

where $\mathrm{sel_P}(\nu) = \{\pi_1 <_{\mathrm{lex}} \cdots <_{\mathrm{lex}} \pi_k\}$ Furthermore, let

$$s^\varphi \xrightarrow{\mathsf{i},\varepsilon}_\nu u^\chi \xrightarrow{\mathsf{i},p_1}_{\rho_1} t_1^{\psi_1} \xrightarrow{\mathsf{i},p_2}_{\rho_2} \cdots \xrightarrow{\mathsf{i},p_n}_{\rho_n} (t_n^{\psi_n}, \Psi)$$

be a $\overset{\mathsf{i}}{\mapsto}_{\mathrm{D_T} \cup \mathcal{R}}$-derivation such that $\rho_j \in \rho_{\mathrm{ass}}$ for some $\rho \in \mathrm{D_T}$, and the p_j are all positions in P_π. Let $Q_\pi = \{i \in \mathbb{N} \mid \pi_i > \pi\}$. Then there exists a π, ρ-narrowing ν_{nar} of ν such that

$$s^\varphi \xrightarrow{\mathsf{i}}_{\nu_{\mathsf{nar}}} u^\chi = \textsc{Collapse}_\varepsilon(\mathrm{filt}_{\varepsilon, Q_\pi}(t_n^{\psi'}))$$

is also derivation for some labelling ψ'.

For the labellings of u^{χ} and $u^{\chi'} = \textsc{Collapse}_{\varepsilon}(\mathrm{filt}_{\varepsilon,Q_\pi}(t_n^{\psi_n}$ holds the following: If $\chi\tau = \Box$ then $\chi'\tau = \Box$ and if $\chi\tau = \chi\tau'$ then also $\chi'\tau = \chi'\tau'$. So the labelling of u^{χ} allows all derivations the labelling of $u^{\chi'}$ allows.

Definition 4.53 (Touched labels)**.** We say that a derivation *touches a label* w, if a step $s^{\varphi} \to [\pi]t^{\psi}$ occurs for any labeled terms s^{φ}, t^{ψ} such that $\varphi(\pi) = w$.

The set of touched labels of a derivation D is denoted by $\mathrm{tl}(D)$.

For the narrowing proof, we need a notation for the length of a concrete derivation sequence.

Definition 4.54. Let $D := t_0 \rightharpoonup_{\rho_1} \cdots \rightharpoonup_{\rho_n} t_n$ be a rewrite sequence and $\rightharpoonup$ one of our rewrite relations ($\to$, $\xrightarrow{\mathsf{i}}$, $\overset{\mathsf{i}}{\mapsto}$, ...). Then the S-length of D is defined to be n, the number of $\rightharpoonup_S$-steps in D. This is denoted by $\mathrm{len}_S(D)$.

Theorem 4.55 (Narrowing)**.** *Let* $(\mathrm{D\textsc{t}}, \mathcal{R})$ *be an annotated CDT problem and* $\nu \in \mathrm{D\textsc{t}}$*. Let* $\nu_1, \ldots, \nu_m$ *be the narrowings with substitutions* $\mu_1, \ldots, \mu_m$ *of a* ν *such that* $s\mu_1, \ldots, s\mu_m$ *are in normal form. If* $\mathrm{sel_T}(\nu) = p \to q$*, let*

$$\nu_0 := (p \to [], (\mathrm{sel_B}(\nu), \mathrm{sel_S}(\nu), \varnothing)).$$

Let $\mathrm{D\textsc{t}}' := \mathrm{D\textsc{t}}[\nu/\{\nu_0, \ldots, \nu_m\}]$ *and* $\mathcal{R}' := \mathcal{R} \cup \{\mathrm{sel_B}(\nu_1), \ldots, \mathrm{sel_B}(\nu_m)\}$*. Then we have* $\mathrm{dl}(t^{\sharp\varepsilon}, \overset{\mathsf{i}}{\mapsto}_{\mathrm{D\textsc{t}}/\mathcal{R}}) \leq 2 \cdot \mathrm{dl}(t^{\sharp\varepsilon}, \overset{\mathsf{i}}{\mapsto} \mathrm{D\textsc{t}}'/\mathcal{R}')$ *for all basic terms* $t^{\sharp} \in \mathcal{T}_B^{\sharp}(\Sigma, \mathcal{V})$*.*

The definition of narrowing for dependency pairs requires that the right-hand side of d does not unify with any left-hand side of a pair. We do not need this restriction here, as rewrite the tuple symbol in this case.

Proof. Let D be a $\overset{\mathsf{i}}{\mapsto}_{\mathrm{D\textsc{t}}\cup\mathcal{R}}$-derivation of $t^{\sharp\varepsilon}$. We apply Lemma 4.56 to replace all $\nu^{\sharp}$-steps. We call the resulting derivation D' and have

$$\mathrm{len}_{\mathrm{D\textsc{t}}}(D) \leq 2 \cdot \mathrm{len}_{\mathrm{D\textsc{t}}'}(D').$$

This proves the theorem. □

The following lemma uses the notation of Theorem 4.55.

Lemma 4.56. *If* D *is a* $\overset{\mathsf{i}}{\mapsto}_{\mathrm{D\textsc{t}}\cup\mathcal{R}}$*-derivation of a correctly corlab term* $(s^{\varphi}, \triangle)$*, then there exists a* $\overset{\mathsf{i}}{\mapsto}_{\mathrm{D\textsc{t}}'\cup\mathcal{R}'}$*-derivation* D_{trans} *of* $(s^{\varphi}, \triangle)$*, with the following properties:*

(1) the first term of D *equals the first term of* D_{trans} *and*

(2) $len_{DT}(D) \leq 2 \cdot len_{\mathrm{D\textsc{t}}'}(D_{\mathsf{trans}})$

Proof (Sketch). We conduct the proof by induction on the number n of $\nu^\sharp$-steps in D whose reduct is not in normal form.

By Lemma 4.35 we may assume that D consists of (potentially infinite) homogeneously labeled subsequence

$$D = D_{w_1}, D_{w_2}, \dots$$

where the $\mathrm{tl}(D_{w_i}) = w_i$ and all w_i are pairwise distinct. Therefore D_{w_i} does not depend on the history of the earlier derivations.

If $n = 0$, D already fulfills the conditions for D_{trans}, so the base case of the induction is trivial. Else, let i be the smallest number such that D_{w_i} contains a ν-step. If $i > 1$, then $D_{w_1}, \dots D_{w_{i-1}}$ already have the form required by D_{trans}. Hence, we may assume $i = 0$.

By the definition of correctly corlab we know that there is at most one tuple position labeled w_1 in the first term in D_{w_1} and therefore there is at most one tuple step in each D_{w_1}. As all positions with the same label are independent, we assume by Lemma 4.33 that

$$D_{w_1} = s_1^{\varphi_1} \xrightarrow{\mathrm{i},p_1}_{\mathrm{sel_B}(\nu)} \cdots \xrightarrow{\mathrm{i},p_{k-1}}_{\mathrm{sel_B}(\nu)} s_k^{\varphi_k} \xrightarrow{\mathrm{i},p_k}_{\nu} s_{k+1}^{\varphi_{k+1}} = t^{\psi}$$

Assume that there is no DT-reduction step below p_k in $D_{\mathsf{rem}} = D_{w_2}, D_{w_3}, \dots$. As all symbols above p_k are compound symbols, there will also never be a derivation at or above p_k in D_{rem}. So we may replace ν by ν_0 and hence $s_{k+1}^{\varphi_{k+1}}$ by $u^{\chi} := t^{\psi}[\mathrm{COM}]_{p_k}$. We call this modified derivation D'_{w_1}. The same replacement must be also be applied to every term in the remaining derivation. If there are $\mathcal{R}$-steps below p_k, we just drop them. We denote the change remaining sequence by $D'_{\mathsf{rem}} := D_{w_2}[\mathrm{COM}]_{p_k}, D_{w_3}[\mathrm{COM}]_{p_k}, \dots$. After this replacement, $\mathrm{len}_{\mathrm{DT}'}(D'_{w_1}) = \mathrm{len}_{\mathrm{DT}}(D_{w_1})$ as there are no DT-steps below p_k and D'_{w_1} does not contain any ν-step anymore. Furthermore u^{χ} is correctly corlab by Lemma 4.26. So we iteratively apply this lemma to D'_{rem}.

Otherwise, there is a DT-reduction step below p_k in D_{rem}. By applying Lemma 4.33 first and then Lemma 4.35 to D_{rem}, we may assume that such a step occurs in D_{w_2}.

Therefore, each subterm labeled with w_2 is at a position below some p_j for $1 \le j \le k$. There is a $\rho \in \mathrm{DT}$ such that all rewrite steps in D_{w_2} are performed with ρ or $\mathrm{sel_B}(\rho)$. As all p_j are independent, we apply Lemma 4.33 to D_{w_1}, D_{w_2} to get a derivation $D' = D'_1, \dots, D'_k$ where a ν_{ass}-step is immediately followed by the ρ_{ass}-steps below and the ν-step is the last ν_{ass}-step:

$$D_{w_1} = s_1^{\varphi_1} \xrightarrow{\mathrm{i},p_1}_{\mathrm{sel_B}(\nu)} s'^{\varphi'_1}_1 \xrightarrow{\mathrm{i},>p_1}_{\mathrm{sel_B}(\rho)} s_2^{\varphi_2} \xrightarrow{\mathrm{i},p_2}_{\mathrm{sel_B}(\nu)} \cdots$$
$$\xrightarrow{\mathrm{i},p_{k-1}}_{\mathrm{sel_B}(\rho)} s_k^{\varphi_k} \xrightarrow{\mathrm{i},p_k}_{\nu} s_{k+1}^{\varphi_{k+1}} \xrightarrow{\mathrm{i},>p_k}{}^{*}_{\mathrm{sel_B}(\rho)} \cdots \xrightarrow{\mathrm{i},p_k.l}_{\rho} = s'^{\varphi'_{k+1}}_{k+1} = t^{\psi}$$

Here we assumed that we have ρ_{ass}-steps below every p_j. Why we can do this without loss of generality is described in the proof of 4.35. So no position labeled with w_2 remains in t^{ψ}.

The l in the last derivation step above selects a right-hand side of ρ and hence is associated with a base position of ρ. Let $Q \subseteq \{1, \cdots, r\}$ be the set of indexes of $\mathrm{sel_P}(\nu) = \{\pi_1 <_{\mathrm{lex}} \cdots <_{\mathrm{lex}} \pi_r\}$, such that $\pi_i > \pi_l$ for all $i \in Q$. As the arguments

$s_{k+1}|_{p_k.l}$ are in normal form (else the ρ-step would not be innermost), also $s_{k+1}|_{p_k.i}$ is in normal form for all $i \in Q$. Therefore, we can filter those positions, without making the derivation shorter. That is, $\mathrm{filt}_{p_k,Q}(t^\psi)$ admits the same derivations as t^ψ and hence $\textsc{Collapse}_{p_k,l}(\mathrm{filt}_{p_k,Q}(t^\psi))$ admits the same derivations. $\textsc{Collapse}_{p_k,l}(\mathrm{filt}_{p_k,Q}(t^\psi)$ admits the same derivations. So, if we apply this transformation to every term in $D_{\mathsf{rem}} := D_{w_3}, D_{w_4}, \cdots$ – we denote this by $D'_{\mathsf{rem}} := \textsc{Collapse}_{p_k,l}(\mathrm{filt}_{p_k,Q}(D_{w_3}, D_{w_4}, \cdots))$ – then D'_{rem} is still a valid derivation, of the same Dt-length.

By Remarks 4.51 and 4.52, we have

$$s_1^{\varphi_1} \xrightarrow{\mathsf{i},p_1}_{\mathrm{sel_B}(\delta)} \cdots \xrightarrow{\mathsf{i},p_{k-1}}_{\mathrm{sel_B}(\delta)} s_k^{\chi_k} \xrightarrow{\mathsf{i},p_k}_{\delta} u^\chi$$

where $u = \textsc{Collapse}_{p_k,l}(\mathrm{filt}_{p_k,Q}(s_{k+1}))$ and by relabeling u^χ admits the same derivations as $\textsc{Collapse}_{p_k,l}(\mathrm{filt}_{p_k,Q}(s_{k+1}^{\varphi_{k+1}}))$.

So again, we can apply this lemma iteratively to D'_{rem}. □

5. Evaluation

Apart from the narrowing processor for annotated CDT problems, all techniques presented here have been implemented in our termination prover AProVE [GST06]. To efficiently find polynomial orders, we used an encoding to propositional logic as described in [FGMSTZ07]. To solve these constraints, we employed the state-of-the-art SAT solver MiniSAT[1]. For approximation of the dependency graph, CAP function and usable rules we used the techniques described in [GTS05a; Thi07]. For the Reduction Pair processor, we also used Usable Rules with regard to argument filtering systems.

As a test bed we employed the current version of the *Termination Problem Database* [Tpd]. This database contains 1381 TRS which are used to test tools analyzing innermost runtime complexity (RCi). In the RCi category of the Termination Competition 2009 a subset of 403 examples was chosen, based on a categorization of those examples in families of similar examples. These 1381 examples are the same as used in the full termination category (i.e. without a restricted rewrite strategy). As such, a significant part of these examples are not good example problems for RCi: Many of them are trivial if one restricts the analysis to basic start terms. In Appendix A we present two small techniques for identifying these systems.

We compared our implementation with the two participants of the RCi category in last year's Termination Competition[2], TCT2[3] and CaT[4]. We will refer to this as *complexity competition*. Our implementation was tested on a 4-core machine with Intel Xeon 5140 processors at 2.33 GHz and 16 GB RAM. CaT and TCT2 ran on the machine used for the termination competition. This machine features eight dual-core AMD Opterons at 2.6GHz and 64GB RAM. For all tools, a timeout of 60 seconds was used.

We employed Theorem A.4 described in Appendix A to find examples with a constant runtime complexity. We proved termination with a very simple method: We used a simple rewriter, which computes all derivations of start terms (and the system is terminating, if and only if the rewriter terminates). This method could prove a constant runtime complexity for 404 problems. In the result tables, we show both the number of successfully solved examples of the whole example collection (TPDB) and the number of those examples not having constant runtime complexity (INT).

In addition, we used AProVE to check for non-termination in these examples. As innermost non-termination does not necessarily imply non-termination on basic terms, we checked some of the non-termination certificates by hand and found that at least 67 of the remaining examples are also innermost non-terminating on start terms. In

[1] http://minisat.se
[2] http://termcomp.uibk.ac.at/status/rules.html
[3] `http://cl-informatik.uibk.ac.at/software/tct/`
[4] `http://cl-informatik.uibk.ac.at/software/cat/`

	DL5	RL5	DQ3	RQ3	DC1	RC1	RA	R+T	TCT2	CaT
TPDB	252	537	312	591	270	589	600	633	189	323
INT	149	175	197	225	189	225	234	251	129	154
COMP	78	168	93	179	80	181	184	198	60	103

Table 5.1.: Experimental results.

Table 5 we give results for three different sets of examples: In the set TPDB are all 1381 examples from the database, INT and COMP are subsets. The set INT contains only those examples which cannot be proved to be of constant runtime complexity by the simple technique above. Finally, COMP contains the 403 examples used in last year's complexity competition.

For our prover we used the strategy outlined in Section 3.4. We only used the transformations from Section 3.3 in the columns labeled $R+T$. In the columns labeled with D, we replaced the reduction pair processor by a technique solving the remaining system with a single reduction pair. In the columns *L5*, *Q3* respectively *C1* we only used a linear, quadratic respectively cubic polynomial interpretations with coefficients up to 5, 3 or 1. For the columns RA and $R+T$, we tried the reduction pair processor with all three orders in parallel. For $R+T$ we allowed up to 6 applications of the transformation techniques of Section 3.3 and gave the reduction pair processor a time limit of 8 seconds.

The strategies described above are for CDT problems. We also implemented the annotated CDT transformation and the order described in Theorem 4.40. Although not described, here we implemented techniques equivalent to the RemoveTrailRhsP and RhsSplitP processor and a technique removing leading and trailing nodes under the conditions described in Lemma 3.25. The narrowing processor was not yet implemented. Only 6 of the examples we could solve with these techniques could not be solved with the $R+T$ strategy; but there are a number of examples for which annotated CDTs proved better (i.e., linear) bounds.

TCT2 mainly employed techniques based on Weak Dependency Pairs, in particular the Weak Dependency Graph. The orders used were Matrix Orders (Arctic matrices [KW08] as well as matrices based on natural numbers), Strongly Linear Interpretations [HM08a] and Polynomial Path Orders. The CaT tool used mainly Matchbounds [KM09], but also matrix interpretations occur in the proofs.

As can be seen from Table 5, our method performs much better than the techniques employed by CaT and TCT2. In particular, we can prove both constant and linear complexity for more examples than can be solved at all with these tools. If we employ all our techniques, this is a factor of 1.9 for the TPDB and COMP sets and a factor of 1.6 for INT. Note that we could prove more examples to be of linear complexity then the other tools could prove at all. Despite these very good results, our technique does not solve a superset of the techniques implemented by the others. On the whole TPDB, of the examples which cannot be solved by CDTs, there are 6 examples which can be solved by TCT2 and 46 examples which can be solved by CaT. The comparison with TCT2 is

particularly interesting, as TCT2 implements another variant for Dependency Pairs for complexity analysis. So experimental evidence suggests that our method is much more powerful on many examples.

Looking at the whole database, we can solve 45% of all examples. If we restrict ourselves to the subset of interesting examples, i.e., those which are not trivially proved to be of constant complexity or to be non-terminating, we can still solve 27.5%.

6. Conclusion

In this thesis we laid down the foundations of a modular and extensible framework for automated complexity analysis of term rewrite systems, in the spirit of the venerable Dependency Pair approach. All results have been successfully automated in the AProVE termination tool. Using this implementation, we were able to prove 633 of 1381 examples respectively 251 of 977 interesting examples to be of at most cubic runtime complexity. These results are better by a wide margin than the results of the winner of the complexity category in the Termination Competition 2009 [1].

Apart from pure numbers, how does our approach compare to the Weak Dependency Pair approach presented by Hirokawa and Moser [HM08a; HM08b]? Both WDPs and CDTs build upon the Dependency Pair approach for termination, but different techniques where chosen to guarantee that the upper bound derived for the transformed system is also an upper bound for the original TRS. Consider the following rule from Example 2.11:

$$\mathsf{quot}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{s}(\mathsf{quot}(\mathsf{minus}(x, y), \mathsf{s}(y)))$$

The classic Dependency Pair transformation generates the pairs

$$\mathsf{QUOT}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{QUOT}(\mathsf{minus}(x, y), \mathsf{s}(y))$$
$$\mathsf{QUOT}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{MINUS}(x, y), \mathsf{s}(y).$$

Our approach generates the same pairs, but ties them together into one right hand side

$$\mathsf{QUOT}(\mathsf{s}(x), \mathsf{s}(y)) \to [\mathsf{QUOT}(\mathsf{minus}(x, y), \mathsf{s}(y)),\ \mathsf{MINUS}(x, y), \mathsf{s}(y)],$$

while for Weak DPs no explicit pair for the minus function call on the right hand side is generated:

$$\mathsf{QUOT}(\mathsf{s}(x), \mathsf{s}(y)) \to \mathsf{QUOT}(\mathsf{minus}(x, y), \mathsf{s}(y))$$

The advantage compared to our approach is that there is no duplication of function symbols. Hence their transformation does not change the complexity of the system and there is no need for a technique similar to our lockstepping. But this advantage comes at a great cost: For Weak Dependency Pairs it is necessary to orient all rules strictly, as evaluation steps both with rules and with pairs have to be counted. To allow the use of an reduction pair which orients only the pairs $\mathcal{P}$ strictly and the rules weakly, the rules have to be compatible with a Strongly Linear Interpretation (SLI) and must be non-duplicating. The latter is a very strong restriction.

[1] http://termcomp.uibk.ac.at/status/rules.html

In contrast to this, the constraints required by our reduction pair processor are very similar to those required by the reduction pair processor for termination: Besides the requirement that one must be able to derive a bound for the number of strict rewrite steps, the only additional restriction is the monotonicity for the compound symbol, which can be easily fulfilled for many orders.

Also, our Complexity Dependency Graph is very similar to the original Dependency Graph and contains a node for each function call in the system. Due to this similarity, many of the powerful techniques developed for termination analysis can be easily adapted to for complexity analysis, as was demonstrated in this thesis. We have seen, that this is not as easy for the annotated variant, so there is still room for improvement. In contrast to that, due to the definition of WDPs, the Weak Dependency Graph makes only the topmost function calls explicit. This prevents techniques like the Rewriting and Narrowing transformations, as those assume that rule steps are only needed to establish a connection between two pair. But for Weak DPs they need to be counted to derive a valid complexity bound. A technique working on the Weak Dependency Graph is *Path Detection* [HM08b]. Unfortunately, the number of paths which have to be considered for this technique may be exponential in the number of nodes and can still be quadratic if the graph of SCCs is a tree. Still, this technique requires SLIs and non-duplicating rules to be able to orient some rules weakly.

A weakness of both CDTs and WDPs is the missing ability to completely remove tuples respectively pairs in SCCs. Our SccSplitP processor is a first step in this direction. In summary one can say that our work greatly enhances the possibilities to automatically derive feasible complexity bounds for Term Rewrite Systems.

6.1. Future Work

We presented a modular framework for the analysis of innermost runtime complexity. It will be interesting to see what is needed to extended the approach presented here to other rewrite strategies and other sets of start terms, e.g. to derivational complexity.

The annotated CDT transformation is complexity preserving, but the annotation of the CDTs makes transformation techniques like narrowing considerably more complicated. We have yet to see how the rewriting technique can be adapted. So work is needed to see whether it is possible to make the annotations optional on a per-tuple basis to restore the modularity of the ordinary CDT approach. As the narrowing for annotated CDTs takes lockstepping into account, it should be considered if it is advantageous to start with annotated CDTs and switch to normal CDTs later, if the annotated method cannot prove a complexity bound. Also, finding a more natural representation for lockstepping would be greatly appreciated. A first step towards this would be to define the complexity in terms of t-chains and chain trees as for unannotated CDT problems. The representation chosen by Hirokawa and Moser [HM08a] for their Weak Dependency Pairs avoids this problem, but loses a great amount of flexibility, as only the topmost function calls are made explicit.

Most transformation theorems presented here do not only yield an upper bound, but

also a lower bound on the complexity. It should be an easy task to extend the framework to support those bounds, too. This would allow analyzing non-termination of TRSs on basic terms. Known techniques for analyzing non-termination [Opp08; GTS05b; TGS08] could be easily adapted, as they produce a certificate of the loop found. For analysis of non-infinite lower bounds, there are no techniques the author knows of yet.

We have chosen to produce only asymptotic complexity results in our framework. For some use cases, it might be useful to be able to compute concrete bounds. To do this, one would need to propagate the complexity values for known nodes down to the leaves of the proof tree, so they can be used to compute the bounds depending on them. On the same matter, we currently compute the result of a processor always by summing up all complexities. It might be useful to allow more flexibility here; this will be required for the technique outlined in the next paragraph.

While our framework allows for a great amount of modularity, it still lacks a technique for analyzing each SCC by itself: We always need to consider all nodes before a SCC, too (albeit in a weakened way). To allow otherwise, one will need to compute bounds on the length (of the relevant part) of the output of nodes. Such an analysis can be strengthened by taking into account sorts, sufficient definedness or the symbols matched by later nodes.

An obvious place for improvement is the addition of additional orders. As we need to choose the interpretation in a way which establishes an upper bound for the complexity, the need for polynomials with a degree higher then 2 is greater than for termination analysis. But with the current state-of-the-art techniques like encoding constraints for the polynomial in propositional logic [FGMSTZ07], the search for such polynomials is often not feasible.

The search for higher-dimensional matrix orders is faster then for polynomials of the same degree and can also be used for complexity analysis [PSS97]. It seems that the constraints on the shape and the values on the diagonal can be lifted for defined function symbols when analyzing runtime complexity. Another candidate is the polynomial path order presented in [AM08]. For best results, the orders must be adapted to be usable as reduction pairs. Avanzini and Moser [AM09] show how to do this for the polynomial path order and Weak Dependency Pairs [HM08a], which have a similar notion of safe reduction pairs as used in this thesis.

It may be advantageous to label nodes by the SCC to which they belong. This would allow different interpretations for the same function symbol in different SCCs. To this end, it might be useful to consider that for polynomial orders we may allow a controlled size increase between SCCs (and probably something similar for other orders).

We have adapted the classic transformation techniques [GTSF06], but not yet the heuristics for their application. Also it should be possible to implement positional narrowing [Thi07], a strong refinement of the standard narrowing technique. Even though we already use strong estimations of Cap and usable rules, those techniques can be improved as we do analysis only on basic start terms. This can be done by keeping track of the symbols which can occur in a term. Narrowing and Rewriting require the system to be weakly innermost terminating (that is, each term can be rewritten to a normal form) to be complete. But of course a complexity proof implies termination, so if we

are able to successfully complete the proof, then we know that the system is weakly innermost terminating (on basic terms) and hence the proof is complete. To use this in the framework, an asymmetric definition of the complexity like in [GTS05a] is needed.

A. Techniques for plain TRSs

The techniques presented in the main part of thesis all work on CDT problems. In this chapter, we present two simple techniques which work on plain Term Rewrite Systems instead.

A.1. Identifying constant runtime complexity

With the techniques presented, we can only identify constant runtime complexity if we get a dependency graph without cycles (without ever using the reduction pair processor). But there is another class of term rewrite systems for which a constant complexity can be easily shown: Systems having only a finite number of (relevant) basic terms.

Throughout this section, $\mathcal{R}$ designates a finite set of rules over alphabet Σ and variables $\mathcal{V}$. With $\rightharpoonup$ we denote either $\to$ or $\xrightarrow{\mathsf{i}}$, depending on whether we talk about full or innermost runtime complexity.

Lemma A.1. *If $\mathcal{T}(\Sigma, \mathcal{V})$ contains only finitely many basic terms and $\mathcal{R}$ is (innermost) terminating on the set of start terms $\mathcal{T}_B$, then $\mathcal{R}$ has constant (innermost) runtime complexity, i.e.,* $\mathrm{rc}(n, \rightharpoonup) \leq c$ *for some* $c \in \mathbb{N}$.

Proof. $\{\mathrm{dl}(t, \rightharpoonup) \mid t \in \mathcal{T}_B(\Sigma, \mathcal{V})\}$ is a finite subset of $\mathbb{N}$ and hence has a maximum. □

Obviously, if all constructor symbols are constants, the set of basic terms can only be finite. Now, the usual assumption is that $\mathcal{V}$ is infinite, so any system containing a non-constant function symbol would have infinite many basic terms. But it is easy see that we may restrict us to basic terms with only one distinct variable symbol. The reason for this is that matching can only detect equality of variables, not inequality.

Lemma A.2. *Let $s \in \mathcal{T}(\Sigma, \mathcal{V})$ and $V \subset \mathcal{V}$ be the set of variables in s. Let $x \in \mathcal{V}$ be a variable and σ a substitution where $v\sigma = x$ for all $v \in V$. Then* $\mathrm{dl}(s, \rightharpoonup) \leq \mathrm{dl}(s\sigma, \rightharpoonup)$.

Proof. Let t be a term and $\pi \in \mathrm{Pos}(t)$. If $s|_\pi$ matches l (for any $l \to r \in \mathcal{R}$), then $s\sigma|_\pi$ matches l. □

In addition, with a similar argument, we may restrict us to constructor symbols occurring on the left-hand side of rules in $\mathcal{R}$: A function symbol not occurring on a left-hand side can only be matched by variables.

Lemma A.3. *Let $s \in \mathcal{T}(\Sigma, \mathcal{V})$. Let $x \in \mathcal{V}$ be a variable and σ be the substitution replacing x by $t = f(t_1, \ldots t_n)$ and t a basic term with f not occurring on the left-hand side of a rule in $\mathcal{R}$. Then* $\mathrm{dl}(s, \rightharpoonup) = \mathrm{dl}(s\sigma, \rightharpoonup)$.

So, we arrive at the following simple theorem:

Theorem A.4. *Let $\mathcal{C}_l$ be the set of of constructor symbols occurring on the left-hand side of rules in $\mathcal{R}$. Let*

$$T := \{f(t_1, \ldots, t_n) \mid f \in \mathcal{D}_\mathcal{R}, t_i \in \mathcal{T}(\mathcal{C}_l, \{x\})\}.$$

If T is finite and $\mathcal{R}$ is (innermost) terminating on T, then $\mathcal{R}$ has constant (innermost) runtime complexity. T is finite if and only iff all constructor symbols are constants.

While such systems may appear to be not very common, a significant amount of examples in the runtime complexity categories of the current Termination Problem Database [Tpd] belongs to this class of problems: This is due to the fact that those examples were constructed for unrestricted start terms instead of basic terms. Of 1381 examples, 432 have only finitely many relevant start terms and 343 of these can be proved to be innermost terminating with a current version of AProVE. It is worth noting that those systems are terminating on all inputs, not only on basic terms. These numbers can be significantly improved if we can teach AProVE to do local termination.

A.2. Start term transformation

By definition, runtime complexity is a start term problem. For the generic problem of local termination of TRS, not many results are known. A recent paper by Endrullis, Vrijer, and Waldmann [EVW09] employs partial algebras for a direct termination method and combine this with semantic labeling to convert a local termination problem into a global termination problem. We describe a simple application of Usable Rules (see 2.4) to remove rules not reachable from start terms. Due to the simple structure of start terms, this approach gives good results.

Remark A.5. Let $\mathcal{R}$ be a set of rules and $\mathcal{R}' \subseteq \mathcal{R}$ be a set containing all rules reachable from a start term, i.e: $\rho \in \mathcal{R}'$ if there exists a basic term s such that $s \to_\mathcal{R} t \to_\rho v$. Then $\mathrm{rc}(n, \to_\mathcal{R}) = \mathrm{rc}(n, \to_{\mathcal{R}'})$.

This allows us to use any over-approximation of usable rules to simplify our initial TRS. Using the usable rules technique implemented in AProVE, this technique is applicable to a set of 723 of the 1381 examples above. If we apply both techniques, we get 463 systems with finite set of (relevant) constructor systems. 435 of them are locally innermost terminating and can be proved as such by AProVE. So at least 31% of the whole example set have constant innermost runtime complexity. One should note, that this technique is not complete; i.e. if the transformed system is nonterminating, the original system might still be terminating on start terms.

Bibliography

[AG00] Thomas Arts and Jürgen Giesl. "Termination of term rewriting using dependency pairs". In: *Theoretical Computer Science* 236.1-2 (2000), pp. 133 –178.

[AM08] Martin Avanzini and Georg Moser. "Complexity Analysis by Rewriting". In: *Proceedings of the 9th International Symposium on Functional and Logic Programming*. Vol. 4989. Lecture Notes in Computer Science. Springer, 2008, pp. 130–146.

[AM09] Martin Avanzini and Georg Moser. "Dependency Pairs and Polynomial Path Orders". In: *Proceedings of the 20th International Conference on Rewriting Techniques and Applications*. Vol. 5595. Lecture Notes in Computer Science. Springer, 2009, pp. 48–62.

[BCMT99a] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet. "Algorithms With Polynomial Interpretation Termination Proof". In: *Journal of Functional Programming* 11 (1999), p. 2001.

[BCMT99b] Guillaume Bonfante, Adam Cichon, Jean-Yves Marion, and Hélène Touzet. "Complexity Classes and Rewrite Systems with Polynomial Interpretation". In: *Proceedings of the 12th International Workshop on Computer Science Logic*. Springer, 1999, pp. 372–384.

[BD86] Leo Bachmair and Nachum Dershowitz. "Commutation, Transformation, and Termination." In: *Proceedings of the 8th International Conference on Automated Deduction*. Vol. 230. Lecture Notes in Computer Science. Springer, 1986, pp. 5–20.

[BN98] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[CL92] Adam Cichon and Pierre Lescanne. "Polynomial Interpretations and the Complexity of Algorithms". In: *Proceedings of the 11th International Conference on Automated Deduction: Automated Deduction*. Springer, 1992, pp. 139–147.

[CMTU05] Evelyne Contejean, Claude Marché, Ana Paula Tomás, and Xavier Urbain. "Mechanically Proving Termination Using Polynomial Interpretations". In: *Journal of Automated Reasoning* 34.4 (2005), pp. 325–363.

[Dbla] *Rewriting Techniques and Applications, 19th International Conference, RTA 2008, Hagenberg, Austria, July 15-17, 2008, Proceedings*. Vol. 5117. Lecture Notes in Computer Science. Springer, 2008.

[Dblb] *Rewriting Techniques and Applications, 20th International Conference, RTA 2009, Brasília, Brazil, June 29 - July 1, 2009, Proceedings*. Vol. 5595. Lecture Notes in Computer Science. Springer, 2009.

[Der87] Nachum Dershowitz. "Termination of rewriting". In: *Journal of Symbolic Computation* 3 (1-2 1987), pp. 69–116.

[EVW09] Jörg Endrullis, Roel C. de Vrijer, and Johannes Waldmann. "Local Termination". In: *Proceedings of the 20th International Conference on Rewriting Techniques and Applications*. Vol. 5595. Lecture Notes in Computer Science. Springer, 2009, pp. 270–284.

[FGMSTZ07] Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. "SAT Solving for Termination Analysis with Polynomial Interpretations". In: *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing*. Vol. 4501. Lecture Notes in Computer Science. Springer, 2007, pp. 340–354.

[Gra96] Bernhard Gramlich. "Termination and Confluence Properties of Structured Rewrite Systems". Fachbereich Informatik, Universität Kaiserslautern, Jan. 1996, p. 217.

[GSST06] Jürgen Giesl, Stephan Swiderski, Peter Schneider-Kamp, and René Thiemann. "Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages". In: *Proceedings of the 17th International Conference on Rewriting Techniques and Applications*. Vol. 4098. Lecture Notes in Computer Science. Springer, 2006, pp. 297–312.

[GST06] Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. "AProVE 1.2: Automatic Termination Proofs in the Dependency Pair Framework". In: *Proceedings of the 3rd International Joint Conference on Automated Reasoning*. Vol. 4130. Lecture Notes in Artificial Intelligence. Springer, 2006, pp. 281–286.

[GTS05a] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. "Proving and Disproving Termination in the Dependency Pair Framework". In: *Deduction and Applications*. Vol. 05431. Dagstuhl Seminar Proceedings. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[GTS05b] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. "Proving and Disproving Termination of Higher-Order Functions". In: *Proceedings of 5th International Workshop on Frontiers of Combining Systems*. Springer, 2005, pp. 216–231.

[GTSF06] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. "Mechanizing and Improving Dependency Pairs". In: *Journal of Automated Reasoning* 37.3 (2006), pp. 155–203.

[HL89] Dieter Hofbauer and Clemens Lautemann. "Termination proofs and the length of derivations". In: *Proceedings of the 3rd international conference on Rewriting Techniques and Applications*. Springer, 1989, pp. 167–177.

[HM08a] Nao Hirokawa and Georg Moser. "Automated Complexity Analysis Based on the Dependency Pair Method". In: *Proceedings of the 4th International Joint Conference on Automated Reasoning*. Vol. 5195. Lecture Notes in Computer Science. Springer, 2008, pp. 364–379.

[HM08b] Nao Hirokawa and Georg Moser. "Complexity, Graphs, and the Dependency Pair Method". In: *Proceedings of the 15th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. Vol. 5330. Lecture Notes in Computer Science. Springer, 2008, pp. 652–666.

[KB70] Donald E. Knuth and Peter B. Bendix. "Simple word problems in universal algebras". In: *Computational Problems in Abstract Algebra*. Pergamon, 1970, pp. 263–267.

[KM09] Martin Korp and Aart Middeldorp. "Match-bounds revisited". In: *Information and Computation* 207.11 (2009), pp. 1259–1283.

[KW08] Adam Koprowski and Johannes Waldmann. "Arctic Termination ...Below Zero". In: *Proceedings of the 19th International Conference on Rewriting Techniques and Applications*. Vol. 5117. Lecture Notes in Computer Science. Springer, 2008, pp. 202–216.

[Lan79] Dallas S. Lankford. *On Proving Term Rewriting Systems are Noetherian*. Memo MTP-3. Revised October 1979. Mathematics Department, Louisiana Tech. University, May 1979.

[Mid01] Aart Middeldorp. "Approximating Dependency Graphs Using Tree Automata Techniques". In: *Proceedings of the First International Joint Conference on Automated Reasoning*. Springer, 2001, pp. 593–610.

[MSW08] Georg Moser, Andreas Schnabl, and Johannes Waldmann. "Complexity Analysis of Term Rewriting Based on Matrix and Context Dependent Interpretations". In: *Proceedings of the IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science*. Vol. 2. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2008, pp. 304–315.

[OBEG10] Carsten Otto, Marc Brockschmidt, Christian von Essen, and Jürgen Giesl. "Automated Termination Analysis of Java Bytecode by Term Rewriting". In: *Proceedings of the 21th International Conference on Rewriting Techniques and Applications (to appear)*. 2010.

[Ohl02] Enno Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002.

[Opp08] Martin Oppelt. "Automatische Erkennung von Ableitungsmustern in nichtterminierenden Wortersetzungssystemen". Diplomarbeit. HTWK Leipzig, July 2008.

[PSS97] Sven Eric Panitz and Manfred Schmidt-Schauß. "TEA: Automatically Proving Termination of Programs in a Non-strict Higher-Order Functional Language". In: *Proceedings of the 4th International Symposium on Static Analysis*. Springer, 1997, pp. 345–360.

[SGST09] Peter Schneider-Kamp, Jürgen Giesl, Alexander Serebrenik, and René Thiemann. "Automated Termination Proofs for Logic Programs by Term Rewriting". In: *Transactions of Computational Logic* (2009).

[Tai61] William W. Tait. "Nested Recursion". In: *Mathematische Annalen* 143 (1961), pp. 236–250.

[TGS08] René Thiemann, Jürgen Giesl, and Peter Schneider-Kamp. "Deciding Innermost Loops". In: *Proceedings of the 19th International Conference on Rewriting Techniques and Applications*. Vol. 5117. Lecture Notes in Computer Science. Springer, 2008, pp. 366–380.

[Thi07] René Thiemann. *The DP Framework for Proving Termination of Term Rewriting*. Tech. rep. AIB-2007-17. RWTH Aachen University, Oct. 2007.

[Tpd] *Termination Problem Database 7.0.2*. URL: `http://www.termination-portal.org/wiki/TPDB`.

[Wal09] Johannes Waldmann. "Automatic Termination". In: *Proceedings of the 20th International Conference on Rewriting Techniques and Applications*. Vol. 5595. Lecture Notes in Computer Science. Springer, 2009, pp. 1–16.

[Wei95] Andreas Weiermann. "Termination Proofs for Term Rewriting Systems by Lexicographic Path Orderings Imply Multiply Recursive Derivation Lengths". In: *Theoretical Computer Science* 139.1&2 (1995), pp. 355–362.

[Zan95] Hans Zantema. "Termination of Term Rewriting by Semantic Labelling". In: *Fundamenta Informaticae* 24.1/2 (1995), pp. 89–105.

[ZHM09] Harald Zankl, Nao Hirokawa, and Aart Middeldorp. "KBO Orientability". In: *Journal of Automated Reasoning* 43.2 (2009), pp. 173–201.