Faculty of Mathematics, Computer Science and Natural Sciences Research Group Computer Science 2

RWTH Aachen University, Germany

Bachelor Thesis

A Term Encoding to Analyze Runtime Complexity via Innermost Runtime Complexity

submitted by

Simeon Valchev

28.03.2024

Reviewers
Prof. Dr. Jürgen Giesl
apl. Prof. Dr. Thomas Noll

Supervisor
Stefan Dollase, M.Sc.

Zentrales Prüfungsamt/Central Examination Office

Eidesstattliche Versicherung
Statutory Declaration in Lieu of an Oath

Name, Vorname/Last Name, First Name

Matrikelnummer (freiwillige Angabe)
Matriculation No. (optional)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel
I hereby declare in lieu of an oath that I have completed the present paper/Bachelor thesis/Master thesis* entitled

selbstständig und ohne unzulässige fremde Hilfe (insbes. akademisches Ghostwriting) erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
independently and without illegitimate assistance from third parties (such as academic ghostwriters). I have used no other than the specified sources and aids. In case that the thesis is additionally submitted in an electronic format, I declare that the written and electronic versions are fully identical. The thesis has not been submitted to any examination body in this, or similar, form.

Ort, Datum/City, Date

Unterschrift/Signature

*Nichtzutreffendes bitte streichen

*Please delete as appropriate

Belehrung:
Official Notification:

§ 156 StGB: Falsche Versicherung an Eides Statt
Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.
Para. 156 StGB (German Criminal Code): False Statutory Declarations
Whoever before a public authority competent to administer statutory declarations falsely makes such a declaration or falsely testifies while referring to such a declaration shall be liable to imprisonment not exceeding three years or a fine.
§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt
(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.
(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.
Para. 161 StGB (German Criminal Code): False Statutory Declarations Due to Negligence
(1) If a person commits one of the offences listed in sections 154 through 156 negligently the penalty shall be imprisonment not exceeding one year or a fine.
(2) The offender shall be exempt from liability if he or she corrects their false testimony in time. The provisions of section 158 (2) and (3) shall apply accordingly.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:
I have read and understood the above official notification:

Ort, Datum/City, Date

Unterschrift/Signature

Abstract

The automation of complexity analysis of term rewrite systems (TRSs) is quite a difficult problem. The tools currently in use to analyze full runtime complexity of TRS can be further improved, as is shown in the results of the Termination Competition. Comparatively, analyzing innermost runtime complexity is stronger, creating the opportunity to use its techniques to infer upper bounds on full runtime complexity. A TRS can be encoded into a new relative TRS, such that its innermost runtime complexity equals the full runtime complexity of the original, allowing us to use these stronger techniques for a different type of analysis. Our approach was formalized, partially proven and implemented. It was tested manually to show an improvement in the full runtime complexity analysis of some examples.

Contents

1. Introduction

Term rewrite systems(TRSs) offer a theoretical foundation to which computer programs can be abstracted and analyzed. One of the topics of research concerning TRSs has to deal with the worst-case lengths of rewrite sequences, more commonly known as complexity. In its analysis a distinction is made based on the starting term of the rewrite sequence: *derivational complexity*, which considers any term, and *runtime complexity*, which considers only basic terms. The more intuitive notion is for a function to be called with some data objects as inputs, which is analogous to the evaluation of basic terms, and is what we focus on. A further distinction is made according to the evaluation strategy. There is *innermost* runtime complexity (irc), whose eager strategy never evaluates a term before its subterms are in normal form, i.e. can no longer be evaluated, and *full* runtime complexity (rc), which utilizes any strategy.

Most of the work in the field has been focused on irc, which is closer to the evaluation of imperative programs. While the topic of rc may not be as well studied, some notable papers on it are [1, 2, 3] with the most recent being [4], which introduced a novel technique for inferring upper bounds for rc by analyzing irc. However, there is still a wide gap in the automatic complexity analysis on rc as can be observed in the results of the annual *Termination Competition* [5]. Of the 959 TRSs analyzed for rc, an upper bound was discovered by at least one of the participants in only 347 cases (36%).

The technique in [4] identifies some TRSs as non-dup generalized (ndg) and shows they have the property $\mathrm{rc}_{\mathcal{R}} = \mathrm{irc}_{\mathcal{R}}$. This combined with the much more powerful analysis of irc allows inferring upper bounds on rc from upper bounds on irc. All non-ndg TRSs are ignored by this method, leaving a gap, that this thesis aims to fill. This is done by extending the technique from [4] by encoding non-ndg TRSs in a way that $\mathrm{rc}_{\mathcal{R}} = \mathrm{irc}'_{\Phi(\mathcal{R})}$, where $\Phi(\mathcal{R})$ is the result of the encoding of $\mathcal{R}$ and irc' refers to irc with a restriction to the starting terms. The two main problems in defining the encoding are calculating the set of non-ndg positions in a TRS and determining if the addition of non-terminating rules is necessary. The latter will make more sense when we dive deeper into the subject. Both of these problems are tackled in this thesis but also leave room for improvement in terms of precision. A remaining problem not discussed here is the analysis of the encoded TRS is confined to the starting terms of the original TRS.

The structure of the thesis is as follows. Chapter 2 introduces some of the preliminary knowledge regarding TRSs. Chapter 3 deals with the problem of calculating the non-ndg positions of a TRS. These are later encoded in our procedure, presented in in the same chapter, which also includes a proof of its usefulness. Chapter 4 introduces a new technique to improve the overall encoding and its analysis by limiting the introduction of non-terminating rules. Chapter 5 serves as a summary of the thesis and a look at future work.

2. Preliminaries

This chapter introduces some of the necessary groundwork on term rewrite systems with accompanying examples. For now, a term can be viewed as some object. Rewriting is then simply a transformation of one term into another. This rewriting is guided by rules, which describe what the input and output of the transformation are. Rules are also associated with some cost, which is considered when analyzing runtime complexity. Unless stated otherwise all rules have a cost of one. Rules can also have conditions besides what input they take, however, such conditional rules are not the focus of this thesis. A term rewrite system is a set of rules.

Example 1. *Consider the TRS $\mathcal{R}_1$ for the calculation of addition on natural numbers.*

$$\begin{array}{lcl} \alpha_1 : \mathsf{add}(x, \mathsf{0}) & \to & x \\ \alpha_2 : \mathsf{add}(x, \mathsf{s}(y)) & \to & \mathsf{add}(\mathsf{s}(x), y) \end{array}$$

Numbers here are represented by the so-called successor function and the symbol 0*, i.e.* 1 *would be* $\mathsf{s}(\mathsf{0})$*,* 2 *would be* $\mathsf{s}(\mathsf{s}(\mathsf{0}))$ *and so on.*

The base case in Ex. 1 would be represented in α_1, where $x+0 = x$. Rule α_2 then recursively subtracts 1 from the second position and adds 1 to the first position. This repeats until the second position reaches 0 and thus the first position is output. One could intuitively imagine this as follows:

$$\begin{array}{ccccccc} 3+2 & = & 4+1 & = & 5+0 & = & 5 \\ \mathsf{add}(\mathsf{s}^3(\mathsf{0}), \mathsf{s}^2(\mathsf{0})) & \to_{\mathcal{R}_1} & \mathsf{add}(\mathsf{s}^4(\mathsf{0}), \mathsf{s}(\mathsf{0})) & \to_{\mathcal{R}_1} & \mathsf{add}(\mathsf{s}^5(\mathsf{0}), \mathsf{0}) & \to_{\mathcal{R}_1} & \mathsf{s}^5(\mathsf{0}) \end{array}$$

To enhance readability, we denote $\mathsf{s}^n(x)$, when the symbol s appears n-many times before some term x. While not explicitly stated in the example, x and y are variables which can be instantiated with other terms. Without a proper definition, one could assume they are some constants like $\mathsf{0}$. Therefore each TRS is associated with a set, which holds all function symbols.

Definition 2 (Signature [6])**.** *A **signature** Σ is a set of function symbols. The number of inputs n of each function symbol is referred to as its arity and it cannot be negative. The subset $\Sigma^{(n)} \subseteq \Sigma$ holds all symbols with arity n. Symbols with arity* 0 *are called constant.*

The signature of $\mathcal{R}_1$ is therefore $\{\mathsf{add}, \mathsf{s}, \mathsf{0}\}$ with the accompanying $\Sigma^{(2)} = \{\mathsf{add}\}$, $\Sigma^{(1)} = \{\mathsf{s}\}$ and $\Sigma^{(0)} = \{\mathsf{0}\}$, whereas x and y are variables.

Definition 3 (Terms [6])**.** *Let Σ be a signature and $\mathcal{V}$ a set of variables such that $\Sigma \cap \mathcal{V} = \emptyset$. The set $\mathcal{T}(\Sigma, \mathcal{V})$ of all **terms** over Σ and $\mathcal{V}$ is the smallest set such that:*

- $\mathcal{V} \subseteq \mathcal{T}(\Sigma, \mathcal{V})$
- *for all $n \geq 0$, all $f \in \Sigma^{(n)}$ and all $t_1, ..., t_n \in \mathcal{T}(\Sigma, \mathcal{V})$, we have $f(t_1, ..., t_n) \in \mathcal{T}(\Sigma, \mathcal{V})$*

The first item states that each variable on its own is a term and the second item specifies that for arbitrary n many terms $t_1, ..., t_n$ and a function symbol f with arity n, $f(t_1, ..., t_n)$ is also a term. $\mathcal{T}(\Sigma, \mathcal{V})$ is denoted as $\mathcal{T}$, if Σ and $\mathcal{V}$ are irrelevant or clear from the context [4].

Given the signature $\Sigma = \{\mathsf{add}, \mathsf{s}, \mathsf{0}\}$ of $\mathcal{R}_1$, we can construct terms like $\mathsf{s}(\mathsf{0})$ or $\mathsf{add}(x, y)$ or more complex ones like $\mathsf{add}(\mathsf{add}(x, \mathsf{s}(\mathsf{s}(\mathsf{0}))), \mathsf{s}(\mathsf{0}))$.

Definition 4 (Term positions [6, 4])**.**

1. *Let Σ be a signature, $\mathcal{V}$ be a set of variables with $\Sigma \cap \mathcal{V} = \emptyset$ and s some term in $\mathcal{T}(\Sigma, \mathcal{V})$. The set of* ***positions*** *of s contains sequences of natural numbers, denoted* $\mathrm{pos}(s)$ *and inductively defined as follows:*

   - *If $s = x \in \mathcal{V}$, then* $\mathrm{pos}(s) := \{\varepsilon\}$
   - *If $s = f(s_1, ..., s_n)$, then*

$$\mathrm{pos}(s) := \{\varepsilon\} \cup \bigcup_{i=1}^{n} \{i.\pi \mid \pi \in \mathrm{pos}(s_i)\}$$

   The symbol ε is used to point to the ***root position*** *of a term. The function symbol at that position is called the* ***root symbol*** *denoted by* $\mathrm{root}(s)$*. Positions can be compared to each other*

$$\pi \leq \tau, \text{ iff there exists } \pi' \text{ such that } \pi.\pi' = \tau$$

   Positions for which neither $\pi \leq \tau$, nor $\pi \geq \tau$ hold, are called ***parallel positions*** *denoted by $\pi \parallel \tau$.*

2. *The* ***size*** *of a term s is defined as $|s| := |\mathrm{pos}(s)|$.*

3. *For $\pi \in pos(s)$, the* ***subterm of*** *s* ***at position*** *π denoted by $s|_\pi$ is defined as*

$$\begin{aligned} s|_\varepsilon &:= s \\ f(s_1, ..., s_n)|_{i.\pi} &:= s_i|_\pi \end{aligned}$$

   If $\pi \neq \varepsilon$, then $s|_\pi$ is a ***proper subterm*** *of s.*

4. *For $\pi \in pos(s)$, the* ***replacement in*** *s* ***at position*** *π* ***with term*** *t denoted by $s[t]_\pi$ is defined as*

$$\begin{aligned} s[t]_\varepsilon &:= t \\ f(s_1, ..., s_n)[t]_{i.\pi} &:= f(s_1, ..., s_i[t]_\pi, ..., s_n) \end{aligned}$$

5. *The set of* ***variables occurring in*** *s, denoted $\mathcal{V}ar(s)$ is defined as*

$$\mathcal{V}ar(s) := \{x \in \mathcal{V} \mid \pi \in \mathrm{pos}(s) \text{ such that } s|_\pi = x\}$$

Consider the term $t = \mathsf{add}(x, \mathsf{s}(y))$. The set of positions is $\mathrm{pos}(t) = \{\varepsilon, 1, 2, 2.1\}$. Some statements that hold for example are $\varepsilon \leq 2 \leq 2.1$ and also $1 \parallel 2$. Some expressions to illustrate are $t|_2 = \mathsf{s}(y)$ and $t[s(0)]_1 = \mathsf{add}(\mathsf{s}(0), \mathsf{s}(y))$.

Definition 5 (Substitution [6])**.** *Let Σ be a signature and $\mathcal{V}$ a finite set of variables. A* ***substitution*** *is a function $\sigma : \mathcal{V} \to \mathcal{T}(\Sigma, \mathcal{V})$ such that $\sigma(x) \neq x$ holds for only finitely many x's. It can be written as*

$$\sigma = \{x_1 \mapsto \sigma(x_1) \ , \ \cdots \ , \ x_n \mapsto \sigma(x_n)\} \ .$$

The term resulting from $\sigma(t)$ is called an ***instance*** *of t. We also write $t\sigma$ instead of $\sigma(t)$.*

For example let $\sigma(x) = \mathsf{s}(\mathsf{0})$ and $t = \mathsf{add}(x, \mathsf{s}(y))$, then $t\sigma = \mathsf{add}(\mathsf{s}(\mathsf{0}), \mathsf{s}(y))$.

Definition 6 (Term rewrite systems [4])**.** *A* ***rewrite rule*** *is a pair of terms $\ell \to r$ such that ℓ is not a variable and $\mathcal{V}ar(r) \subseteq \mathcal{V}ar(\ell)$. It is referred to ℓ as the left-hand side and r as the right-hand side. A* ***term rewrite system*** *is a finite set of such rewrite rules associated with a signature Σ.*

1. *Given the signature* Σ *of a TRS* $\mathcal{R}$ *we have*
   - *the* ***set of defined symbols*** $\Sigma_d := \{\text{root}(\ell) \mid \ell \to r \in \mathcal{R}\}$
   - *the* ***set of constructor symbols*** $\Sigma_c := \Sigma \setminus \Sigma_d$
2. *A term* $f(s_1, ..., s_n)$ *is* ***basic****, if* $f \in \Sigma_d$ *and* $s_1, ..., s_n \in \mathcal{T}(\Sigma_c, \mathcal{V})$*. The* ***set of all basic terms*** *over* Σ *and* $\mathcal{V}$ *is denoted by* $\mathcal{T}_B(\Sigma, \mathcal{V})$*.*
3. *The* ***number of occurrences of*** x ***in term*** t *is denoted by* $\#_x(t)$*.*
4. *A* ***redex*** *(reducible expression) is an instance of* ℓ *of some rule* $\ell \to r$*.*
   - *A term* t *is an* ***innermost redex****, if none of its proper subterms is a redex.*
   - *A term* t *is in* ***normal form****, if none of its subterms is a redex.*

For $\mathcal{R}_1$ in Ex. 1 we have $\Sigma_d = \{\mathsf{add}\}$ and $\Sigma_c = \{\mathsf{0}, \mathsf{s}\}$. Consider the term $s = \mathsf{add}(\mathsf{add}(\mathsf{0}, \mathsf{0}), \mathsf{0})$. With a substitution $\sigma = \{x \mapsto \mathsf{add}(\mathsf{0}, \mathsf{0})\}$ it holds that s is an instance of the left-hand side of rule α_1 and therefore s is a redex. Further, s is not an innermost one as the subterm $s|_1 = \mathsf{add}(\mathsf{0}, \mathsf{0})$ can be reduced to $\mathsf{0}$. Since none of the subterms of $s|_1$ can be reduced, it is an innermost redex.

Definition 7 (Rewrite step [4])**.** *A* ***rewrite step*** *is a reduction(or evaluation) of a term* s *to* t *by applying a rule* $\ell \to r$ *at position* π*, denoted by* $s \to_{\ell \to r, \pi} t$ *and means that for some substitution* σ*,* $s|_\pi = \sigma(\ell)$ *and* $t = s[\sigma(r)]_\pi$ *holds.*

1. *Rule and position in the subscript can be omitted, if they are irrelevant. We denote* $s \to_{\mathcal{R}} t$*, if it holds for some rule in* $\mathcal{R}$*, and in order to distinguish between rules and rewrite steps.*
2. *A sequence of rewrite steps(or rewrite sequence)* $s = t_0 \to_{\mathcal{R}} t_1 \to_{\mathcal{R}} \cdots \to_{\mathcal{R}} t_m = t$ *is denoted by* $s \to^m_{\mathcal{R}} t$*.*
3. *Only rules with cost* 1 *contribute to the length of a rewrite sequence, i.e. for* $s \to^m_{\mathcal{R}} t$ *it holds that* $m = 0$*, if it is a single rewrite step that uses a rule with* 0 *cost.*
4. *A rewrite step is called* ***innermost****, if* $s|_\pi$ *is an innermost redex, and is denoted by* $s \xrightarrow{i}_\pi t$*.*

The term $s = \mathsf{add}(\mathsf{add}(\mathsf{0}, \mathsf{0}), \mathsf{0})$ mentioned above can be used to create the following rewrite sequence in $\mathcal{R}_1$:

$$\mathsf{add}(\mathsf{add}(\mathsf{0}, \mathsf{0}), \mathsf{0}) \to_{\mathcal{R}_1} \mathsf{add}(\mathsf{0}, \mathsf{0}) \to_{\mathcal{R}_1} \mathsf{0}$$

Definition 8 (Derivation height [4])**.** *The* ***derivation height*** $\text{dh} : \mathcal{T} \times 2^{\mathcal{T} \times \mathcal{T}} \to \mathbb{N} \cup \{\omega\}$ *is a function with two inputs: a term* t *and a binary relation on terms, in this case* $\to$*. It defines the length of the longest sequence of rewrite steps starting with term* t*, i.e.*

$$\text{dh}(t, \to) = \sup\{m \mid t' \in \mathcal{T} \ , \ t \to^m t'\}$$

In the case of an infinitely long rewrite sequence starting with term s*, we denote* $\text{dh}(s, \to) = \omega$*.*

Definition 9 ((Innermost) Runtime complexity [4])**.** *The* ***runtime complexity***(rc) *of a TRS* $\mathcal{R}$ *maps any* $n \in \mathbb{N}$ *to the length of the longest* $\to$*-sequence (rewrite sequence) starting with a basic term* t *with* $|t| \leq n$*. The innermost runtime complexity*(irc) *is defined analogously, but it only considers innermost rewrite steps. More precisely* $\text{rc}_{\mathcal{R}} : \mathbb{N} \to \mathbb{N} \cup \{\omega\}$ *and* $\text{irc}_{\mathcal{R}} : \mathbb{N} \to \mathbb{N} \cup \{\omega\}$*, defined as:*

$$\text{rc}_{\mathcal{R}}(n) = \sup\{\text{dh}(t, \to_{\mathcal{R}}) \mid t \in \mathcal{T}_B, |t| \leq n\},$$
$$\text{irc}_{\mathcal{R}}(n) = \sup\{\text{dh}(t, \xrightarrow{i}_{\mathcal{R}}) \mid t \in \mathcal{T}_B, |t| \leq n\}$$

Since every $\xrightarrow{i}$-sequence can be viewed as a $\to$-sequence, it is clear that $\text{irc}_{\mathcal{R}}(n) \leq \text{rc}_{\mathcal{R}}(n)$. Therefore, an upper bound for $\text{rc}_{\mathcal{R}}$ infers an upper bound for $\text{irc}_{\mathcal{R}}$ and a lower bound for $\text{irc}_{\mathcal{R}}$ infers a lower bound for $\text{rc}_{\mathcal{R}}$. For now we can say that the length of rewrite sequences in $\mathcal{R}_1$ depends solely on the second argument of the starting basic term. The size of this subterm is decremented by one after each rewrite until it reaches $\mathsf{0}$, terminating after one more rewrite.

The TRS $\mathcal{R}_1$ also belongs to a class of system, for which full and innermost runtime complexity are equal. This class was the focus of [4] and we aim to go beyond it in this thesis.

Definition 10 (Non-dup-generalized [4, 7])**.** *The rewrite step $s \to_{\ell \to r, \pi} t$ with the matching substitution σ is called* ***non-dup-generalized****(ndg), if*

- *For all variables x with $\#_x(r) > 1$, $\sigma(x)$ is in normal form, and*
- *For all $\tau \in \mathrm{pos}(\ell) \backslash \{\varepsilon\}$ with $\mathrm{root}(\ell|_\tau) \in \Sigma_d$, it holds that $\sigma(\ell|_\tau)$ is in normal form.*

A TRS $\mathcal{R}$ is called ndg, if every rewrite sequence starting with a basic term consists of only ndg rewrite steps. A ***non-ndg*** *rewrite step does not fulfill at least one of the criteria and a TRS is non-ndg if it produces a sequence with a non-ndg rewrite step.*

The first condition regards the duplication of redexes, which obviously affects the runtime complexity and is by definition unreachable via an innermost evaluation strategy. The second condition regards nested redexes whose root symbol is matched. This means a potential rewrite step is ignored, which once again can affect the runtime complexity in unexpected ways.

One can now easily observe that the TRS $\mathcal{R}_1$ is indeed ndg since none of its rules duplicate variables on the right-hand side or have nested defined symbols on the left-hand side. It is impossible for a rewrite sequence starting with a basic term to reach a non-ndg rewrite step. Therefore it holds that $\mathrm{rc}_{\mathcal{R}_1} = \mathrm{irc}_{\mathcal{R}_1}$. However, we are interested in non-ndg TRSs.

Example 11. *Consider the non-ndg TRS $\mathcal{Q}$, which simply counts down given some input:*

$$\begin{aligned}
&\alpha_1 : \mathsf{f}(x) \to \mathsf{g}(\mathsf{a}(x)) && \alpha_2 : \mathsf{a}(x) \to \mathsf{b}(\mathsf{x}) \\
&\alpha_3 : \mathsf{g}(\mathsf{b}(x)) \to \mathsf{lin}(x) && \alpha_4 : \mathsf{g}(\mathsf{a}(x)) \to \mathsf{quad}(x) \\
&\alpha_5 : \mathsf{lin}(\mathsf{s}(x)) \to \mathsf{lin}(x) && \alpha_6 : \mathsf{quad}(\mathsf{s}(x)) \to \mathsf{c}(\mathsf{lin}(x), \mathsf{quad}(x))
\end{aligned}$$

For now, we can assume that the starting term of the longest rewrite sequences has the root symbol f. Innermost evaluation would force the subterm $\mathsf{a}(x)$ to be reduced to $\mathsf{b}(x)$ and thus create a rewrite sequence with length in $\mathcal{O}(n)$. Otherwise, the term can be matched to the left-hand side of α_4, and in a rewrite step that is non-ndg, create a rewrite sequence with length in $\mathcal{O}(n^2)$. In other words, $\mathrm{irc}_{\mathcal{Q}} \in \mathcal{O}(n)$*, while* $\mathrm{rc}_{\mathcal{Q}} \in \mathcal{O}(n^2)$*.*

While a non-ndg TRS $\mathcal{R}$ fails to satisfy $\mathrm{rc}_{\mathcal{R}} = \mathrm{irc}_{\mathcal{R}}$, we show there exists a function Φ, which encodes $\mathcal{R}$ into $\Phi(\mathcal{R})$. This encoded TRS has a very specific property: the innermost runtime complexity of $\Phi(\mathcal{R})$, constrained to the basic terms of $\mathcal{R}$, is equal to the full runtime complexity of $\mathcal{R}$. In short, $\mathrm{rc}_{\mathcal{R}} = \mathrm{irc}'_{\Phi(\mathcal{R})}$, where irc' is the modified irc as described above.

3. Encoding

By definition every TRS $\mathcal{R}$ has finitely many rules and therefore finitely many positions, which can be encoded. However, for practical purposes, encodings that encode every possible position are not useful. Generally speaking, encodings like this introduce new rules, which can produce loops and create infinitely long rewrite sequences albeit using rules with 0 cost. The focus of this chapter lies in the optimization of which positions to encode (Section 3.1.) and the limitation of non-terminating rules (Section 3.2.). To this effort, the goal of the encoding is to ensure each innermost rewrite sequence in $\Phi(\mathcal{R})$ has an equally long full rewrite sequence in $\mathcal{R}$ and vice versa, which then serves as proof of $\mathrm{rc}_{\mathcal{R}} = \mathrm{irc}'_{\mathcal{R}'}$.

Consider again the TRS $\mathcal{Q}$ from Ex. 11 and the rule $\alpha_4 : \mathsf{g}(\mathsf{a}(x)) \to \mathsf{quad}(x)$. We already pointed out that evaluating with this rule creates a non-ndg rewrite step, due to the fact the subterm $\mathsf{a}(x)$ is a redex for any instantiation. However, replacing the defined root symbol with some fresh constructor symbol, makes every rewrite step using that rule innermost. Doing that everywhere in the TRS for a specifically will allow for innermost rewrite sequences with length in $\mathcal{O}(n^2)$. Consider the eventual encoding $\Phi(\mathcal{Q})$:

$$\begin{aligned}
&\alpha'_1 : \mathsf{f}(x) \to \mathsf{g}(\mathsf{i}(l_{\mathsf{a}}(x))) && \alpha'_2 : \mathsf{a}(x) \to \mathsf{b}(\mathsf{x})\\
&\alpha'_3 : \mathsf{g}(\mathsf{b}(x)) \to \mathsf{lin}(x) && \alpha'_4 : \mathsf{g}(l_{\mathsf{a}}(x)) \to \mathsf{quad}(\mathsf{i}(x))\\
&\alpha'_5 : \mathsf{lin}(\mathsf{s}(x)) \to \mathsf{lin}(x) && \alpha'_6 : \mathsf{quad}(\mathsf{s}(x)) \to \mathsf{c}(\mathsf{lin}(\mathsf{i}(x)), \mathsf{quad}(\mathsf{i}(x)))\\
&\beta_1 : \mathsf{i}(x) \xrightarrow{0} x && \beta_2 : \mathsf{i}(l_{\mathsf{a}}(x)) \xrightarrow{0} \mathsf{i}(\mathsf{a}(x))\\
& && \beta_3 : \mathsf{i}(l_{\mathsf{a}}(x)) \xrightarrow{0} \mathsf{i}(l_{\mathsf{a}}(\mathsf{i}(x)))
\end{aligned}$$

While everything we discussed is as we expected, there are certainly more differences than we mentioned. Besides the addition of the fresh constructor symbol for a, there is also a newly defined symbol i, which is at the root of three 0-cost rules, one of which restores the defined symbol a in the place of l_{a}.

As mentioned previously, the goal is not to alter the rules of the TRS so that only its longest rewrite sequence is innermost, but to simulate it and every other rewrite sequence via an innermost strategy. The newly introduced rules combined with the replacing of certain defined symbols with their constructor counterpart, and their subsequent encapsulation by an i-symbol allow us to do so. Consider the innermost rewrite sequence

$$\mathsf{f}(0) \quad \to_{\alpha'_1} \quad \mathsf{g}(\mathsf{i}(l_{\mathsf{a}}(0))) \quad \xrightarrow{0}_{\beta_1} \quad \mathsf{g}(l_{\mathsf{a}}(0)) \quad \to_{\alpha'_4} \quad \mathsf{quad}(\mathsf{i}(0))$$

We can say that it is a simulation of the rewrite sequence in $\mathsf{f}(0) \to_{\mathcal{Q}} \mathsf{g}(\mathsf{a}(0)) \to_{\mathcal{Q}} \mathsf{quad}(0)$. Both sequences have the same cost and for now can be conjectured that this holds for any other rewrite sequence in $\Phi(\mathcal{Q})$, that starts with a basic term in $\mathcal{Q}$. We will now analyze why we need this restriction on starting terms. Consider the rewrite sequence with a starting term that is basic in $\Phi(\mathcal{Q})$:

$$\mathsf{f}(l_{\mathsf{a}}(0)) \quad \to_{\alpha'_1} \quad \mathsf{g}(\mathsf{i}(l_{\mathsf{a}}(l_{\mathsf{a}}(0)))) \quad \xrightarrow{0}{}^{3}_{\Phi(\mathcal{Q})} \quad \mathsf{g}(\mathsf{i}(l_{\mathsf{a}}(\mathsf{a}(0)))) \quad \to_{\alpha'_4} \quad \mathsf{g}(\mathsf{i}(l_{\mathsf{a}}(\mathsf{b}(0)))) \quad \to_{\Phi(\mathcal{Q})} \cdots$$

This is obviously a rewrite sequence that cannot be mapped to a rewrite sequence in $\mathcal{Q}$, i.e. it is a rewrite sequence that we are not interested in. Therefore, if we want useful results from the encoding, we must limit the starting terms to the ones from $\mathcal{Q}$.

But that is not all that we can notice by observing this encoded TRS. There is a duplication of variables in rule α_6, which are now also encapsulated by an i-symbol on the right-hand side. The reason is that the encoding has discovered the potential of a redex to flow into these duplicated variables, which as we remind is also a condition to be avoided for ndg rewriting. We expand on the issue of how these positions in the TRS are determined to be encoded in the next section.

3.1 Non-ndg locations

Before we establish how we tag non-ndg positions, we need to introduce two more definitions to help us better define this flow. The first of these are the so-called locations, which refer to positions in a TRS, so we can more easily tell them apart from the positions in terms.

Definition 12 (Location)**.** *Let $\mathcal{R}$ be a TRS. A **location** in $\mathcal{R}$ is a triple $(\alpha, \mathtt{X}, \pi)$, where $\alpha \in \mathcal{R}$, $\mathtt{X} \in \{\mathtt{L}, \mathtt{R}\}$ and π a position on the left- or right-hand side of α, depending on $\mathtt{X}$. The set of all **locations** of $\mathcal{R}$ is defined as:*

$$\mathcal{L}_\mathcal{R} := \{(\alpha, \mathtt{L}, \pi) \mid \alpha = \ell \to r \in \mathcal{R},\ \pi \in \mathrm{pos}(\ell)\} \cup \{(\alpha, \mathtt{R}, \pi) \mid \alpha = \ell \to r \in \mathcal{R},\ \pi \in \mathrm{pos}(r)\}$$

- *The set of all **sub-locations** of a location λ is defined as*
  $\mathrm{loc}_\mathcal{R}(\lambda) := \{(\alpha, \mathtt{X}, \pi) \mid \lambda = (\alpha, \mathtt{X}, \tau),\ \tau \leq \pi,\ (\alpha, \mathtt{X}, \pi) \in \mathcal{L}_\mathcal{R}\}$,
- *For all $\ell \to r \in \mathcal{R}$ we write $\mathcal{R}|_{(\ell \to r, \mathtt{L}, \pi)} = \ell|_\pi$ and $\mathcal{R}|_{(\ell \to r, \mathtt{R}, \pi)} = r|_\pi$.*

The matching of nested redexes does create some problems, which we will explore further later. One solution to this is temporarily replacing them with fresh variables, while the matching occurs. We do this by using the CAP-function.

Definition 13 ($\mathrm{CAP}_\mathcal{R}$ [8])**.** *For a given TRS $\mathcal{R}$ and some term q we define $\mathrm{CAP}_\mathcal{R} : \mathcal{T} \to \mathcal{T}$ as follows:*

$$\mathrm{CAP}_\mathcal{R}(q) = \begin{cases} q & \text{if} \quad q \in \mathcal{V} \\ f(\mathrm{CAP}_\mathcal{R}(t_1), \ldots, \mathrm{CAP}_\mathcal{R}(t_n)) & \text{else if } q = f(t_1, \ldots t_n), f \in \Sigma_c \\ x & \text{else if } x \text{ is a fresh variable} \end{cases}$$

In other words, in the resulting term $\mathrm{CAP}_\mathcal{R}(q)$ all proper subterms of a term q, which have a defined root symbol, are replaced with different fresh variables. Multiple occurrences of the same subterm are replaced by pairwise different variables.

Consider the TRS $\mathcal{Q}$ from Ex. 11 and the term $q = \mathsf{g}(\mathsf{a}(x))$, which corresponds to the right-hand side of rule α_1. As we will discover later, the location of the a-symbol flows into a non-ndg location in rule α_4. We discover this flow by matching $\mathrm{CAP}_\mathcal{R}(q) = g(x)$ to the left-hand side of α_4. While it would have matched regardless of the application of the $\mathrm{CAP}_\mathcal{R}$-function, there are cases where this is not quite as obvious.

As we continue to the algorithm for tagging non-ndg locations it is important to say that while some work on data flow analysis exists [9], it was deemed not precise enough to use for our encoding. Instead, it was preferred to create a separate algorithm, that would return all to-be-encoded locations in a given TRS. The following set contains all locations tagged as non-ndg.

Definition 14. *Let $\mathcal{R}$ be a TRS.*

1. *The **set of base non-ndg locations** $\mathcal{X}'_\mathcal{R}$ is the smallest set such that:*

   - $\{(\alpha, \mathtt{L}, \tau) \mid \alpha = \ell \to r \in \mathcal{R}, \quad \tau \in \mathrm{pos}(\ell) \backslash \{\varepsilon\},\ \ \mathrm{root}(\ell|_\tau) \in \Sigma_d\} \subseteq \mathcal{X}'_\mathcal{R}$ $(S1)$
   - $\{(\alpha, \mathtt{R}, \tau) \mid \alpha = \ell \to r \in \mathcal{R}, \tau, \pi \in \mathrm{pos}(r), \tau \neq \pi, r|_\tau = r|_\pi \in \mathcal{V}\} \subseteq \mathcal{X}'_\mathcal{R}$ $(S2)$

   The locations in $\mathcal{X}'_\mathcal{R}$ are the ones where a defined symbol is nested on the left-hand side of a rule $(S1)$, and where variables are duplicated on the right-hand side of a rule $(S2)$.

2. *The* ***set of over-approximated non-ndg locations*** $\mathcal{X}_\mathcal{R}$ *is the smallest set such that:*

   - $\mathcal{X}'_\mathcal{R} \subseteq \mathcal{X}_\mathcal{R}$
   - $\left\{(\alpha, \mathtt{L}, \tau) \;\middle|\; \begin{array}{l}\alpha = \ell \to r \in \mathcal{R}, \tau \in \mathrm{pos}(\ell), \pi \in \mathrm{pos}(r),\\ \ell|_\tau = r|_\pi \in \mathcal{V}\end{array}\right\} \subseteq \mathcal{X}_\mathcal{R}$, *if* $(\alpha, \mathtt{R}, \pi) \in \mathcal{X}_\mathcal{R}$ $(S3)$
   - $\left\{(\alpha, \mathtt{R}, \tau) \;\middle|\; \begin{array}{l}\alpha = \ell \to r \in \mathcal{R},\ \tau \in \mathrm{pos}(r), \tau = \pi.\upsilon, \upsilon \neq \varepsilon\\ \beta = s \to t \in \mathcal{R}, w \in \mathrm{pos}(s), \upsilon \nparallel w,\\ \text{MGU } \textit{for } \mathrm{CAP}_\mathcal{R}(r|_\pi) \textit{ and } s \textit{ exists}\end{array}\right\} \subseteq \mathcal{X}_\mathcal{R}$, *if* $(\beta, \mathtt{L}, w) \in \mathcal{X}_\mathcal{R}$ $(S4)$
   - $\left\{(\alpha, \mathtt{R}, \tau) \;\middle|\; \begin{array}{l}\alpha = \ell \to r \in \mathcal{R}, \tau \in \mathrm{pos}(r),\\ \beta = s \to t \in \mathcal{R}, \pi \in \mathrm{pos}(t),\\ \text{MGU } \textit{for } \mathrm{CAP}_\mathcal{R}(t|_\pi) \textit{ and } \ell \textit{ exists}\end{array}\right\} \subseteq \mathcal{X}_\mathcal{R}$, *if* $(\beta, \mathtt{R}, \pi) \in \mathcal{X}_\mathcal{R}$ $(S5)$

   $\mathcal{X}_\mathcal{R}$ includes all base non-ndg locations and then some more based on what flows into these. The first set $(S3)$ includes the locations of variables on the left-hand side of rules, which flow directly into tagged variable locations on the right-hand side of the same rules. Simply put, the tagging of a location of variable x on the right-hand side, tags all locations of x on the left-hand side. The second set $(S4)$ tracks the flow of locations from the right-hand side of rules to the left-hand side of rules. And finally $(S5)$ contains all over-approximated reachable return locations of tagged defined symbols at right-hand side of rules. Return locations of some symbol f *refer to all locations on the right-hand side of a rule, whose left-hand side root symbol is* f.

We can now go into further detail about why each of these locations is tagged as non-ndg. Since our encoding will be analyzed via its irc, all tagged locations of defined symbols on the left-hand side of any rule will be replaced by a corresponding constructor in the encoded TRS. The locations of such subterms are contained in $(S1)$.

Next to consider is the duplication of variables. Or the more relevant - duplication of redexes. This is impossible when employing an innermost evaluation strategy. That is why we must track backward what flows into these variables and encode it, i.e. replace by corresponding constructors. The initial duplicated positions are contained in $(S2)$ and then $(S3)$ tracks back to the left-hand side of the rules in $(S2)$ to tag their origin. Afterward, $(S4)$ checks if any subterm at the right-hand side of any rule can be matched to the tagged locations in $(S1 - S2)$.

Since $(S4 - S5)$ are more complex compared to the rest, we will go into further detail for each step. In $(S4)$ the position τ is split into two parts: $\tau = \pi.\upsilon$, where $\upsilon \neq \varepsilon$. The subterm at π is what we try to match. While $\pi = \varepsilon$ is allowed, $\upsilon = \varepsilon$ is not, because this would include the ε positions of right-hand side of rules, which can not flow between locations. The MGU (Most General Unifier) is the matcher between the two terms and it means that a subterm reached via a rewrite step using rule α, can then potentially be evaluated using rule β.

Notably, we also use the $\mathrm{CAP}_\mathcal{R}$-function on the subterm we match. The reason we use it is to ease the discovery of data flows of redexes, i.e. the location of a redex may flow into a non-ndg location but also fail to match with the term at the said location. Since the redex can be rewritten, so as to actually match the term, we decided to over-approximate and basically assume it can always be rewritten to match. Due to its similarity to the Word problem, this was considered appropriate. The most obvious way to show the over-approximation caused by $\mathrm{CAP}_\mathcal{R}$ is via a simple example.

$$\beta_1 : \mathsf{h} \to \mathsf{g}(\mathsf{h}) \qquad \beta_2 : \mathsf{g}(\mathsf{a}) \to \mathsf{g}(\mathsf{a}) \qquad \beta_3 : \mathsf{a} \to \mathsf{a}$$

Here the h term on the right-hand side of β_1 does not flow into position 1 of the left-hand side of β_2. Using the $\mathrm{CAP}_\mathcal{R}$ function on $\mathsf{g}(\mathsf{h})$ returns $\mathsf{g}(x)$, which can be matched.

Lastly, we require that $\upsilon \nparallel w$. This part ensures that the newly included location in α actually flows into a non-ndg location. Since it may not be as obvious how we arrived at this part of Def. 14, we can consider an example build to show all cases.

$$\alpha_1 : \mathsf{h}(x_1, y_1, z_1) \to \mathsf{g}(x_1, y_1, \mathsf{s}(z_1)) \quad \alpha_2 : \mathsf{g}(x_2, \mathsf{s}(y_2), z_2) \to \mathsf{d}(x_2, x_2)$$

Clearly, rule α_2 is duplicating and the locations of x_2 are all tagged as non-ndg. Next, we investigate α_1, where the variable x_1 on the right-hand side flows into x_2. Their positions are not just non-parallel, they are equal, which was how we defined this set originally. However,

upon further inspection, we also noticed that y_1 should flow into y_2. We can not know what y_1 can be instantiated with, but since it can be for example $\mathsf{s}(t)$ for some term t, then that is what flows between the locations. Then for the position of y_1 and y_2 it holds that $2 \leq 2.1$. Analogously can be derived for the location of z_1, which flows into the location of z_2. Here the entire term $\mathsf{s}(z_1)$ is what flows between locations. Going back to their positions we have $3.1 \geq 3$. Thus we only require that the positions of the locations to be tagged and the already tagged location are non-parallel. If we were not to consider the relation between these positions, then a single tagged location somewhere in the matched term is needed to tag any location in the matcher. As per the example, that would mean tagging y_1 and z_1, which do flow into their respective locations in α_2, because x_2 is tagged.

In $(S5)$ are included the return locations of non-ndg tagged defined symbols. The subterms on the right-hand sides of rules, which have a defined root symbol, maybe redexes and thus rewritten. The newly evaluated term at the position of rewriting must inherit the non-ndg tag, otherwise said term may flow into a non-ndg location, defeating the whole purpose of the encoding.

In terms of computation, $\mathcal{X}_\mathcal{R}$ for some TRS $\mathcal{R}$ has to be continuously reevaluated until no further changes are made. Since no locations are ever removed from the set and there are finitely many locations in a TRS, we can conclude that $\mathcal{X}_\mathcal{R}$ always reaches a fix-point.

Expanding on TRS $\mathcal{Q}$ from Ex. 11, we can now see how the newly defined set helps in establishing the set of locations to be encoded. The locations in $\mathcal{X}_\mathcal{Q}$ are underlined.

$$\alpha_1 : \mathsf{f}(x) \to \mathsf{g}(\underbrace{\mathsf{a}(x)}) \qquad \alpha_2 : \mathsf{a}(\underset{\sim}{x}) \to \underbrace{\mathsf{b}(x)}$$

$$\alpha_3 : \mathsf{g}(\mathsf{b}(x)) \to \mathsf{lin}(x) \qquad \alpha_4 : \mathsf{g}(\underbrace{\mathsf{a}(x)}) \to \mathsf{quad}(\underset{\sim}{x})$$

$$\alpha_5 : \mathsf{lin}(\mathsf{s}(x)) \to \mathsf{lin}(x) \qquad \alpha_6 : \mathsf{quad}(\mathsf{s}(\underset{\sim}{x})) \to \mathsf{c}(\mathsf{lin}(\underset{\sim}{x}), \mathsf{quad}(\underset{\sim}{x}))$$

Besides the variable duplication in α_6 and the nested redex at α_4, there are a few other locations tagged as non-ndg. We previously noticed the flow from α_1 to α_4, but there is also one from α_4 to α_6. Further, the return locations of a in α_2 are tagged. Cross-referencing this tagging with the encoding $\Phi(\mathcal{Q})$ leaves some locations not encoded. All but one are locations of variables on the left-hand side of rules, which will later have to be excluded, as it makes no sense to encode them too. The other one is the location of a constructor symbol and therefore not a redex, so we have no use for encoding that location, as it will not affect the runtime complexity. For the same reason, we would have no use for encoding locations of variable duplication, if the only thing getting duplicated are constructor terms. Assume an extension $\mathcal{Q}_1 = \mathcal{Q} \cup \{\mathsf{d}(x, y) \to \mathsf{d}(x, x)\}$. When calculating $\mathrm{rc}_{\mathcal{Q}_1}$ only basic terms are allowed as starting terms. Therefore the duplication of variable x in the extended rule will always receive some constructor term and would prove useless to encode.

At this point, $\mathcal{X}_\mathcal{Q}$ is sufficient to define an encoding of $\mathcal{Q}$, but as shown in the extended example, we can improve the precision of this over-approximation. Calculating a different set of locations, which tracks the flow of defined symbols on the right-hand sides of rules, will then allow us to define all non-ndg tagged locations, which can potentially receive a redex.

Definition 15. *For a TRS $\mathcal{R}$ and a set of locations Δ let $\mathcal{Y}^\Delta_\mathcal{R}$ be the smallest set containing Δ and all locations, which locations from Δ might flow into:*

- $\Delta \subseteq \mathcal{Y}^\Delta_\mathcal{R}$ $(S6)$
- $\left\{ (\alpha, \mathtt{R}, \pi) \;\middle|\; \begin{array}{l} \alpha = \ell \to r \in \mathcal{R}, \tau \in \mathrm{pos}(\ell), \pi \in \mathrm{pos}(r), \\ \ell|_\tau = r|_\pi \in \mathcal{V} \end{array} \right\} \subseteq \mathcal{Y}^\Delta_\mathcal{R}$, *if* $(\alpha, \mathtt{L}, \tau) \in \mathcal{Y}^\Delta_\mathcal{R}$ $(S7)$
- $\left\{ (\beta, \mathtt{L}, w) \;\middle|\; \begin{array}{l} \alpha = \ell \to r \in \mathcal{R},\ \tau \in \mathrm{pos}(r), \tau = \pi.\upsilon, \upsilon \neq \varepsilon, \\ \beta = s \to t \in \mathcal{R}, w \in \mathrm{pos}(s), \upsilon \nparallel w, \\ \text{MGU } \textit{for } \mathrm{CAP}_\mathcal{R}(r|_\pi) \textit{ and } s \textit{ exists} \end{array} \right\} \subseteq \mathcal{Y}^\Delta_\mathcal{R}$, *if* $(\alpha, \mathtt{R}, \tau) \in \mathcal{Y}^\Delta_\mathcal{R}$ $(S8)$
- $\left\{ (\beta, \mathtt{R}, \pi) \;\middle|\; \begin{array}{l} \alpha = \ell \to r \in \mathcal{R}, \tau \in \mathrm{pos}(r), \\ \beta = s \to t \in \mathcal{R}, \pi \in \mathrm{pos}(t), \\ \text{MGU } \textit{for } \mathrm{CAP}_\mathcal{R}(t|_\pi) \textit{ and } \ell \textit{ exists} \end{array} \right\} \subseteq \mathcal{Y}^\Delta_\mathcal{R}$, *if* $(\alpha, \mathtt{R}, \tau) \in \mathcal{Y}^\Delta_\mathcal{R}$ $(S9)$

The definitions of these subsets are similar to $(S3 - S5)$. Changed is which location from a data flow is included.

Applying this to TRS $\mathcal{Q}$ with $\Delta = \{(\alpha, \mathtt{R}, \tau) \mid \alpha = \ell \to r \in \mathcal{Q}, \mathrm{root}(r|_\tau) \in \Sigma_d\}$ we get

$$\alpha_1 : \mathsf{f}(x) \to \mathsf{g}(\underset{\sim}{\widehat{\mathsf{a}}}(\underset{\sim}{x})) \qquad \alpha_2 : \mathsf{a}(\underset{\sim}{x}) \to \underset{\sim}{\mathsf{b}}(\underset{\sim}{x})$$
$$\alpha_3 : \mathsf{g}(\mathsf{b}(x)) \to \widehat{\mathsf{lin}}(x) \qquad \alpha_4 : \mathsf{g}(\underset{\sim}{\widehat{\mathsf{a}}}(\underset{\sim}{\widehat{x}})) \to \widehat{\mathsf{quad}}(\underset{\sim}{\widehat{x}})$$
$$\alpha_5 : \mathsf{lin}(\mathsf{s}(x)) \to \widehat{\mathsf{lin}}(x) \qquad \alpha_6 : \mathsf{quad}(\mathsf{s}(\underset{\sim}{\widehat{x}})) \to \mathsf{c}(\underset{\sim}{\widehat{\mathsf{lin}}}(\underset{\sim}{\widehat{x}}), \underset{\sim}{\widehat{\mathsf{quad}}}(\underset{\sim}{\widehat{x}}))$$

where the locations in $\mathcal{Y}^{\Delta}_{\mathcal{Q}}$ are indicated with $\widehat{\ }$ (hat) over them. The intersection of underlined locations and locations with a hat, gives us the set of over-approximated non-ndg locations, in which a defined symbol flows. As can be seen, this will ignore the tagged constructor in α_2, as was desired. Further, even though the lin-symbols at α_3 and α_5 have a hat, they do not flow into a non-ndg location, therefore encoding them is not necessary. The locations in that intersection are now exactly the ones encoded in $\Phi(\mathcal{Q})$.

Definition 16 ($\mathrm{ENC}_{\mathcal{R}}$)**.** *For a TRS $\mathcal{R}$ and $\Delta = \{(\alpha, \mathtt{R}, \tau) \mid \alpha = \ell \to r \in \mathcal{R}, \mathrm{root}(r|_\tau) \in \Sigma_d\}$ let the set of all locations to be encoded $\mathrm{ENC}_{\mathcal{R}}$ be defined as*

$$\mathrm{ENC}_{\mathcal{R}} := (\mathcal{X}_{\mathcal{R}} \cap \mathcal{Y}^{\Delta}_{\mathcal{R}}) \setminus \{(\alpha, \mathtt{L}, \pi) \mid \mathcal{R}|_{(\alpha,\mathtt{L},\pi)} \in \mathcal{V}\}$$

In order to show the thought process behind the encoding, we focus on a simpler example. For the purpose of convenience we chose a TRS that only has a few rules and its runtime complexity is constant.

Example 17. *Consider the non-ndg TRS $\mathcal{U}$, which has its locations in $\mathrm{ENC}_{\mathcal{U}}$ underlined.*

$$\alpha_1 : \mathsf{h} \to \mathsf{g}(\underline{\mathsf{a}}, \underline{\mathsf{a}}, \mathsf{a}) \qquad \alpha_2 : \mathsf{g}(x, \underline{\mathsf{a}}, \mathsf{y}) \to \mathsf{f}(\underline{x}, \underline{x}) \qquad \alpha_3 : \mathsf{a} \to \mathsf{b}$$

Since we strive to analyze its irc, the nested a-symbol in α_2 can not stay as is and must be changed to some fresh constructor symbol. We choose l and in order to differentiate between other such constructor symbols, we indicate the original symbol being replaced in the subscript, giving us l_{a}.

$$\alpha_1 : \mathsf{h} \to \mathsf{g}(\underline{\mathsf{a}}, \underline{\mathsf{a}}, \mathsf{a}) \qquad \alpha_2 : \mathsf{g}(x, \underline{l_{\mathsf{a}}}, \mathsf{y}) \to \mathsf{f}(\underline{x}, \underline{x}) \qquad \alpha_3 : \mathsf{a} \to \mathsf{b}$$

However, this change eliminates the flow from α_1 to α_2. If we want to preserve it, we can also encode in the same fashion the a-symbols in α_1.

$$\alpha_1 : \mathsf{h} \to \mathsf{g}(\underline{l_{\mathsf{a}}}, \underline{l_{\mathsf{a}}}, \mathsf{a}) \qquad \alpha_2 : \mathsf{g}(x, \underline{l_{\mathsf{a}}}, \mathsf{y}) \to \mathsf{f}(\underline{x}, \underline{x}) \qquad \alpha_3 : \mathsf{a} \to \mathsf{b}$$

Replacing defined symbols with a constructor creates a new problem. There are rewrite sequences that evaluate these exact terms, that we have now practically removed. In Ex. 17 after rewriting with α_1, only one a term in the left position can be evaluated, in comparison to the three a terms before the change. If we can restore the defined symbol at these positions, then that would not be an issue. Ideally, we do this via a rule with 0 cost, so as to not affect time complexity. This means we need to introduce a new defined symbol and new rules.

$$\alpha_1 : \mathsf{h} \to \mathsf{g}(\underline{\mathsf{i}(l_a)}, \underline{\mathsf{i}(l_a)}, \mathsf{a}) \qquad \alpha_2 : \mathsf{g}(x, \underline{l_{\mathsf{a}}}, \mathsf{y}) \to \mathsf{f}(\underline{x}, \underline{x}) \qquad \alpha_3 : \mathsf{a} \to \mathsf{b}$$
$$\alpha_4 : \mathsf{i}(l_{\mathsf{a}}) \xrightarrow{0} \mathsf{a}$$

Again, we run into a problem of matching the right-hand side of α_1 to left-hand side of α_2. While position 1 can be matched, the rewrite step would not be innermost. And even ignoring that, position 2 would fail to match anyway. In situations like this we would like to remove the i-symbols from the redex, so the rewrite is innermost.

$$\alpha_1 : \mathsf{h} \to \mathsf{g}(\underline{\mathsf{i}(l_a)}, \underline{\mathsf{i}(l_a)}, \mathsf{a}) \qquad \alpha_2 : \mathsf{g}(x, \underline{l_{\mathsf{a}}}, \mathsf{y}) \to \mathsf{f}(\underline{x}, \underline{x}) \qquad \alpha_3 : \mathsf{a} \to \mathsf{b}$$
$$\alpha_4 : \mathsf{i}(l_{\mathsf{a}}) \xrightarrow{0} \mathsf{i}(\mathsf{a}) \qquad \alpha_5 : \mathsf{i}(x) \xrightarrow{0} x$$

After all these changes, the duplication in α_2 has remained unaltered. We will show now why the tagged x's should be encapsulated by i, similar to the right-hand side of α_1. Consider the following rewrite sequence using the rules α_1 to α_5 directly above.

$$\mathsf{h} \quad \to_{\alpha_1} \quad \mathsf{g}(\mathsf{i}(l_{\mathsf{a}}), \mathsf{i}(l_{\mathsf{a}}), \mathsf{a}) \quad \xrightarrow{0}_{\alpha_5} \quad \mathsf{g}(l_{\mathsf{a}}, \mathsf{i}(l_{\mathsf{a}}), \mathsf{a}) \quad \xrightarrow{0}_{\alpha_5} \quad \mathsf{g}(l_{\mathsf{a}}, l_{\mathsf{a}}, \mathsf{a}) \quad \to_{\alpha_2} \quad \mathsf{f}(l_{\mathsf{a}}, l_{\mathsf{a}})$$

The last term $\mathsf{f}(l_\mathsf{a}, l_\mathsf{a})$ cannot be further evaluated, since there is no way to restore the a-symbols from these constructors. We can assume from the concept of the encoding so far that it is possible that redexes flow into these tagged variable positions. Therefore an i-symbol should be placed there when rewriting with α_2.

$$\alpha_1 : \mathsf{h} \to \mathsf{g}(\mathsf{i}(l_\mathsf{a}), \mathsf{i}(l_\mathsf{a}), \mathsf{a}) \qquad \alpha_2 : \mathsf{g}(x, l_\mathsf{a}, y) \to \mathsf{f}(\mathsf{i}(x), \mathsf{i}(x)) \qquad \alpha_3 : \mathsf{a} \to \mathsf{b}$$

$$\alpha_4 : \mathsf{i}(l_\mathsf{a}) \xrightarrow{0} \mathsf{i}(\mathsf{a}) \qquad \alpha_5 : \mathsf{i}(x) \xrightarrow{0} x$$

We get the final version of the TRS, which corresponds to the eventual encoding $\Phi(\mathcal{U})$. So far we discussed and formalized what locations in a TRS are to be encoded. In conclusion, only changing the rules is not sufficient, and new rules should be added. The additional rules exclusively evaluate redexes with the newly introduced defined symbol i at their root and can be split into three categories.

- $\mathsf{i}(l_\mathsf{a}) \xrightarrow{0} \mathsf{i}(\mathsf{a})$

  The *executing rules* restore the defined symbol, whose constructor version is at position 1 on the left-hand side. Shorthand notation for this rule is $\texttt{EXE}_\mathsf{a}$, where a stands for the defined symbol being restored.

- $\mathsf{i}(x) \xrightarrow{0} x$

  The *omission rule* is mandatory in all encodings, as was shown in the previous section. Shorthand notation for this rule is $\texttt{OMIT}$.

- $\mathsf{i}(l_\mathsf{g}(x, y, z)) \xrightarrow{0} \mathsf{i}(l_\mathsf{g}(\mathsf{i}(x), \mathsf{i}(y), \mathsf{i}(z)))$

  The *propagation rule* encapsulates with an i-symbol all subterms directly below the root of the term matched at position 1. This type of rule was missing from Ex. 17 because it was indeed not needed. We know this because g was the only symbol that had non-zero arity but was not encoded. Shorthand notation for this rule is $\texttt{PROP}_\mathsf{g}$, where g is the symbol at which the propagation occurs.

  The propagation rules have two variations:

  - *Terminating* $\quad \mathsf{i}(l_\mathsf{g}(x, y, z)) \xrightarrow{0} l_\mathsf{g}(\mathsf{i}(x), \mathsf{i}(y), \mathsf{i}(z))$
  - *Non-terminating* $\quad \mathsf{i}(l_\mathsf{g}(x, y, z)) \xrightarrow{0} \mathsf{i}(l_\mathsf{g}(\mathsf{i}(x), \mathsf{i}(y), \mathsf{i}(z)))$

  We make this distinction due to the consequences of adding both the omission rule and the propagation rule for any symbol. The non-terminating rule above in combination with $\mathsf{i}(x) \to x$ for some encoded TRS $\Phi(\mathcal{R})$ creates non-terminating rewrite sequences like the following:

$$\cdots \quad \to_{\mathcal{R}'} \quad \mathsf{i}(l_\mathsf{g}(\mathsf{b}, \mathsf{b}, \mathsf{b})) \quad \xrightarrow{0}_{\mathcal{R}'} \quad \mathsf{i}(l_\mathsf{g}(\mathsf{i}(\mathsf{b}), \mathsf{i}(\mathsf{b}), \mathsf{i}(\mathsf{b}))) \quad \xrightarrow{0}{}^*_{\mathcal{R}'} \quad \mathsf{i}(l_\mathsf{g}(\mathsf{b}, \mathsf{b}, \mathsf{b})) \quad \xrightarrow{0}_{\mathcal{R}'} \quad \cdots$$

  It is currently unknown how to analyze the irc of a TRS with non-terminating 0-cost rules. An analysis of the rules for which function symbols it would be viable to add the terminating version of the rule, is provided in the next chapter.

For each of these shorthand notation, we use a superscript L or R to indicate a left-hand side or right-hand side of the respective rule. For example $\texttt{OMIT}^L = \mathsf{i}(x)$.

The propagation rules are required for function symbols, which are encoded and have a non-zero arity. Without further analysis, the encoding would have to add the non-terminating propagation rules for these symbols. As discussed, this is not ideal and we should focus on eliminating as many of these non-terminating rules as possible, which will be covered in the next chapter.

The encoding should also add propagation rules for constructor symbols. This has not been covered in any of the examples so far but is quite trivial, because if there happens to be a location with a constructor symbol below a non-ndg tagged location and another location of an encoded defined symbol below that, then there is no way to propagate the i-symbol inward without the appropriate rules. While a TRS can be further analyzed to give us the necessary constructor symbols, an encoding, that adds propagation rules for all of them, does not in any

meaningful way differ from one that limits it. We can also reason that the propagation rule for any constructor symbol can be terminating without creating any obstacles to the automatic complexity analysis. That is because these terms can never be a redex and therefore there is no need to keep them encapsulated by an i-symbol.

3.2 Formal definition

In this section, we introduce the definition of the encoding, which so far has only been partially conceptualized. As a reminder, the goal of the encoding is to encode a TRS $\mathcal{R}$ to $\Phi(\mathcal{R})$, whose modified innermost runtime complexity is the same as the full runtime complexity of $\mathcal{R}$. Thus allowing us to use the much more powerful techniques for irc analysis to derive results for rc, mainly an upper-bound on rc.

Definition 18. *Let $\mathcal{R}$ be a TRS. We define **the encoding** as a function Φ :*

$$\Phi(\mathcal{R}) = \mathcal{R}' \cup \mathcal{S}, \text{ such that}$$

$$\begin{aligned}
\mathcal{R}' &= \{\psi(\alpha) \mid \alpha \in \mathcal{R}\} \\
\mathcal{S} &= \{\mathsf{i}(x) \xrightarrow{0} x\} \\
&\cup \{\, \mathsf{i}(l_f(x_1,\dots,x_n)) \xrightarrow{0} \mathsf{i}(l_f(\mathsf{i}(x_1),\dots,\mathsf{i}(x_n))) \mid f = \mathrm{root}(\mathcal{R}|_\lambda) \in \Sigma_d^n, \lambda \in \mathrm{ENC}_\mathcal{R}\} \\
&\cup \{\, \mathsf{i}(l_f(x_1,\dots,x_n)) \xrightarrow{0} \mathsf{i}(f(x_1,\dots,x_n)) \mid f = \mathrm{root}(\mathcal{R}|_\lambda) \in \Sigma_d^n, \lambda \in \mathrm{ENC}_\mathcal{R}\} \\
&\cup \{\, \mathsf{i}(d(x_1,\dots,x_n)) \xrightarrow{0} d(\mathsf{i}(x_1),\dots,\mathsf{i}(x_n)) \mid d \in \Sigma_c^n\}
\end{aligned}$$

ψ encodes the rules in $\mathcal{R}$ and uses φ to encode the individual sides of each rule. We define the ψ-function as follows:

$$\psi(\alpha) = \varphi(\mathcal{R}, (\alpha, \mathtt{L}, \varepsilon)) \to \varphi(\mathcal{R}, (\alpha, \mathtt{R}, \varepsilon)), \;\; \alpha \in \mathcal{R}$$

For some location $\lambda = (\alpha, \mathtt{X}, \pi)$, $\mathtt{X} \in \{\mathtt{L}, \mathtt{R}\}$, $k \in \mathbb{N}$ let $\lambda^{(k)} = (\alpha, \mathtt{X}, \pi.k)$, and $f = \mathrm{root}(\mathcal{R}|_\lambda) \in \Sigma^n$ or $f \in \mathcal{V}$.

1. $$\varphi(\mathcal{R}, \lambda) = \begin{cases} f(\;\varphi(\mathcal{R},\lambda^{(1)}),\dots,\;\varphi(\mathcal{R},\lambda^{(n)})) & \text{if} \quad f \in \Sigma_c \text{ or } \lambda \notin \mathrm{ENC}_\mathcal{R} \\ l_f(\;\varphi(\mathcal{R},\lambda^{(1)}),\dots,\;\varphi(\mathcal{R},\lambda^{(n)})) & \text{else if } \mathtt{X} = \mathtt{L} \\ \mathsf{i}\;(l_f(\varphi_\mathcal{N}(\mathcal{R},\lambda^{(1)}),\dots,\varphi_\mathcal{N}(\mathcal{R},\lambda^{(n)}))) & \text{else if } f \notin \mathcal{V} \\ \mathsf{i}(f) & \text{else} \end{cases}$$

2. $$\varphi_\mathcal{N}(\mathcal{R}, \lambda) = \begin{cases} f(\varphi(\mathcal{R},\lambda^{(1)}),\dots,\varphi(\mathcal{R},\lambda^{(n)})) & \text{if} \quad f \in \Sigma_c \\ l_f(\varphi(\mathcal{R},\lambda^{(1)}),\dots,\varphi(\mathcal{R},\lambda^{(n)})) & \text{else if } f \in \Sigma_d \\ f & \text{else} \end{cases}$$

We can now see precisely how a TRS gets encoded via Φ. so an example that incorporates different aspects of the encoding is presented.

Example 19. *Consider the following TRS $\mathcal{P}$ and all the relevant information about its encoding. The locations in $\mathrm{ENC}_\mathcal{P}$ are underlined.*

$$\begin{array}{lll} \alpha_1 : \mathsf{h} \to \mathsf{a}(\underline{\mathsf{f}(\mathsf{c})}) & & \alpha_2 : \mathsf{a}(x) \to \mathsf{a}(\underline{\mathsf{a}(x)}) \\ \alpha_3 : \mathsf{f}(\underline{\mathsf{c}}) \to \underline{\mathsf{f}(\mathsf{c})} & \alpha_4 : \mathsf{a}(x) \to \mathsf{b}(\underline{x}, \underline{x}) & \alpha_5 : \mathsf{c} \to \mathsf{0} \end{array}$$

We are now going to take a look at how $\Phi(\mathcal{P})$ is calculated by going through each step. Terms with zero non-ndg tagged locations are skipped since the first condition of φ clearly ignores these and changes nothing in the process.

First we encode each rule to get $\mathcal{P}'$:

$\psi(\alpha_1) = \mathsf{h} \to \varphi(\mathsf{a}(\mathsf{f}(\mathsf{c}))) \quad = \mathsf{h} \to \mathsf{a}(\varphi(\mathsf{f}(\mathsf{c}))) \quad = \mathsf{h} \to \mathsf{a}(\mathsf{i}(l_\mathsf{f}(\varphi_\mathcal{N}(\mathsf{c})))) \quad = \mathsf{h} \to \mathsf{a}(\mathsf{i}(l_\mathsf{f}(l_\mathsf{c})))$

$\psi(\alpha_2) = \mathsf{a}(x) \to \varphi(\mathsf{a}(\mathsf{a}(x))) = \mathsf{a}(x) \to \mathsf{a}(\varphi(\mathsf{a}(x))) = \mathsf{a}(x) \to \mathsf{a}(\mathsf{i}(l_\mathsf{a}(\varphi_\mathcal{N}(x)))) = \mathsf{a}(x) \to \mathsf{a}(\mathsf{i}(l_\mathsf{a}(x)))$

$\psi(\alpha_3) = \varphi(\mathsf{f}(\mathsf{c})) \to \varphi(\mathsf{f}(\mathsf{c})) \;= \mathsf{f}(\varphi(\mathsf{c})) \to \mathsf{i}(l_\mathsf{f}(\varphi_\mathcal{N}(\mathsf{c}))) = \mathsf{f}(l_\mathsf{c}) \to \mathsf{i}(l_\mathsf{f}(l_\mathsf{c}))$

$\psi(\alpha_4) = \mathsf{a}(x) \to \varphi(\mathsf{b}(x,x)) \;= \mathsf{a}(x) \to \mathsf{b}(\varphi(x), \varphi(x)) \;= \mathsf{a}(x) \to \mathsf{b}(\mathsf{i}(x), \mathsf{i}(x))$

$\psi(\alpha_5) = \alpha_5$

We summarize the process of encoding the rules of $\mathcal{P}$. The subterm $\mathsf{f}(\mathsf{c})$ on the right-hand side of α_1 flows into the duplication of α_4 and thus is encapsulated by an i-symbol, analogous to the subterm $\mathsf{a}(x)$ in α_2. The only case of a nested defined symbol on the left-hand side of a rule in TRS $\mathcal{P}$ is located in α_3. It is only replaced by a fresh constructor symbol and the entire right-hand side of α_3 is encoded, since these are return locations of f, tagged non-ndg in α_1. The right-hand side of α_4 contains the duplicated variables, also encoded to $\mathsf{i}(x)$. Next come the additional rules. Instead of grouping them by category, where all `PROP` rules and all `EXE` rules are combined in one set, it is much more readable and easy to understand if we group them by function symbols. We start with the `OMIT` rule, which is not tied to any function symbol. Next, we iterate over the symbols which are underlined above, i.e. which get encoded. Let the first symbol be f. The encoding adds $\mathtt{EXE}_\mathsf{f}$ and the non-terminating $\mathtt{PROP}_\mathsf{f}$.

- $\mathcal{S} \supseteq \{\mathsf{i}(x) \xrightarrow{0} x\} \cup \{\mathsf{i}(l_\mathsf{f}(x)) \xrightarrow{0} \mathsf{i}(\mathsf{f}(x)),\ \mathsf{i}(l_\mathsf{f}(x)) \xrightarrow{0} \mathsf{i}(l_\mathsf{f}(\mathsf{i}(x)))\}$

Next symbol we pick is c. It has an arity of 0, therefore no $\mathtt{PROP}_\mathsf{c}$ rule can be added. Thus only $\mathtt{EXE}_\mathsf{c}$ is included in $\mathcal{S}$.

- $\mathcal{S} \supseteq \{\mathsf{i}(x) \xrightarrow{0} x\} \cup \{\mathsf{i}(l_\mathsf{f}(x)) \xrightarrow{0} \mathsf{i}(\mathsf{f}(x)),\ \mathsf{i}(l_\mathsf{f}(x)) \xrightarrow{0} \mathsf{i}(l_\mathsf{f}(\mathsf{i}(x)))\} \cup \{\mathsf{i}(l_\mathsf{c}) \xrightarrow{0} \mathsf{i}(\mathsf{c})\}$

The last function symbol to consider is a. with a step similar to the one for f.

- $\mathcal{S} \supseteq \{\mathsf{i}(x) \xrightarrow{0} x\} \cup \{\mathsf{i}(l_\mathsf{f}(x)) \xrightarrow{0} \mathsf{i}(\mathsf{f}(x)),\ \mathsf{i}(l_\mathsf{f}(x)) \xrightarrow{0} \mathsf{i}(l_\mathsf{f}(\mathsf{i}(x)))\} \cup \{\mathsf{i}(l_\mathsf{c}) \xrightarrow{0} \mathsf{i}(\mathsf{c})\}$
  $\cup \{\mathsf{i}(l_\mathsf{a}(x)) \xrightarrow{0} \mathsf{i}(\mathsf{a}(x)),\ \mathsf{i}(l_\mathsf{a}(x)) \xrightarrow{0} \mathsf{i}(l_\mathsf{a}(\mathsf{i}(x)))\}$

By Def. 18 we should also add propagation rules for constructor symbols, but that was not precisely demonstrated previously and only mentioned in passing. In TRS $\mathcal{P}$ there is only one constructor symbol with non-zero arity - b. It is not tagged as non-ndg, since locations with constructor root symbols were excluded from $\mathrm{ENC}_\mathcal{P}$. Sparing any further analysis on the TRS, we can simply add the terminating rules for all constructor symbols, with which $\mathcal{S}$ is finally complete.

- $\mathcal{S} = \{\mathsf{i}(x) \xrightarrow{0} x\} \cup \{\mathsf{i}(l_\mathsf{f}(x)) \xrightarrow{0} \mathsf{i}(\mathsf{f}(x)),\ \mathsf{i}(l_\mathsf{f}(x)) \xrightarrow{0} \mathsf{i}(\mathsf{l}_\mathsf{f}(\mathsf{i}(x)))\} \cup \{\mathsf{i}(l_\mathsf{c}) \xrightarrow{0} \mathsf{i}(\mathsf{c})\}$
  $\cup \{\mathsf{i}(l_\mathsf{a}(x)) \xrightarrow{0} \mathsf{i}(\mathsf{a}(x)),\ \mathsf{i}(l_\mathsf{a}(x)) \xrightarrow{0} \mathsf{i}(l_\mathsf{a}(\mathsf{i}(x)))\} \cup \{\mathsf{i}(\mathsf{b}(x,x) \xrightarrow{0} \mathsf{b}(\mathsf{i}(x), \mathsf{i}(x))\}$

With these results we can construct the full encoding $\Phi(\mathcal{P})$

$\alpha'_1 : \mathsf{h} \to \mathsf{a}(\mathsf{i}(l_\mathsf{f}(l_\mathsf{c})))$		$\alpha'_2 : \mathsf{a}(x) \to \mathsf{a}(\mathsf{i}(l_\mathsf{a}(x)))$
$\alpha'_3 : \mathsf{f}(l_\mathsf{c}) \to \mathsf{i}(l_\mathsf{f}(l_\mathsf{c}))$	$\alpha'_4 : \mathsf{a}(x) \to \mathsf{b}(\mathsf{i}(x), \mathsf{i}(x))$	$\alpha'_5 : \mathsf{c} \to \mathsf{0}$
$\beta_1 : \mathsf{i}(x) \xrightarrow{0} x$	$\beta_2 : \mathsf{i}(l_\mathsf{f}(x)) \xrightarrow{0} \mathsf{i}(\mathsf{f}(x))$	$\beta_3 : \mathsf{i}(l_\mathsf{f}(x)) \xrightarrow{0} \mathsf{i}(l_\mathsf{f}(\mathsf{i}(x)))$
	$\beta_5 : \mathsf{i}(l_\mathsf{a}(x)) \xrightarrow{0} \mathsf{i}(\mathsf{a}(x))$	$\beta_6 : \mathsf{i}(l_\mathsf{a}(x)) \xrightarrow{0} \mathsf{i}(l_\mathsf{a}(\mathsf{i}(x)))$
	$\beta_4 : \mathsf{i}(l_\mathsf{c}) \xrightarrow{0} \mathsf{i}(\mathsf{c})$	$\beta_7 : \mathsf{i}(\mathsf{b}(x,x) \xrightarrow{0} \mathsf{b}(\mathsf{i}(x), \mathsf{i}(x))$

One thing that is important to clarify is that the resulting TRS is not ndg. When we talk about expanding the technique from [4], we do not mean transforming the TRS into an equivalent ndg TRS, for which irc = rc holds. We can show this via a simple rewrite sequence:

$$\mathsf{h} \quad \to_{\alpha'_1} \quad \mathsf{a}(\mathsf{i}(l_\mathsf{f}(l_\mathsf{c}))) \quad \to_{\alpha'_4} \quad \mathsf{b}(\mathsf{i}^2(l_\mathsf{f}(l_\mathsf{c})), \mathsf{i}^2(l_\mathsf{f}(l_\mathsf{c})))$$

What we see in the second rewrite step is a duplication of a redex, namely the one at position 1. Therefore $\Phi(\mathcal{P})$ is not an ndg TRS, but that is not a problem, since the encoding has not been

designed to eliminate the duplication of redexes. Instead, it was designed so that it could work around the duplication of redexes.

So far it was stated multiple times that the encoding over-approximates. It would be good to also look into an example of a TRS that is ndg, but whose rules still end up getting encoded.

Example 20. *Consider the following TRS $\mathcal{M}$ with locations in* $\text{ENC}_{\mathcal{M}}$ *underlined.*

$$\alpha_1 : \mathsf{f} \to \mathsf{g}(\underline{\mathsf{a}}) \qquad \alpha_2 : \mathsf{a} \to \mathsf{s}(\mathsf{0}) \qquad \alpha_3 : \mathsf{g}(\mathsf{s}(x)) \to \mathsf{c}(\underline{x}, \underline{x})$$

In this example, the reason why the location of a *in* α_1 *is included in* $\text{ENC}_{\mathcal{M}}$ *is that before matching to see if a flow occurs, the* a*-symbol is replaced by a variable through the* $\text{CAP}_{\mathcal{M}}$*-function. In that case, the flow is over-approximated and leads to an encoding of an ndg TRS.*

Next, we show that TRS $\mathcal{M}$ is indeed ndg. Firstly, there are no nested defined symbols on the left-hand side of any rule, so one of the conditions of ndg can never be met. The second is the duplication of redexes. This can only occur when evaluating with α_3 and specifically if a redex flows into the variable x. For that matter, we need to consider all other occurrences of g in $\mathcal{M}$. The only other one is in α_1, where the only proper subterm is a redex. As mentioned previously $\mathsf{g}(\mathsf{a})$ can not be matched with the left-hand side of α_3. However, we should continue by exploring the evaluations of a, of which there is only one and is obviously in normal form. Therefore, all rewrite steps in $\mathcal{M}$ are ndg, making $\mathcal{M}$ an ndg TRS, which can be analyzed via the technique in [4] without the usage of the encoding. Regardless, the encoding Φ will not preserve the rules and will consider this a non-ndg TRS.

At this point we can start working towards proving that $\text{rc}_{\mathcal{R}} = \text{irc}'_{\Phi(\mathcal{R})}$ for any TRS $\mathcal{R}$. As a reminder, we measure the $\text{irc}'_{\Phi(\mathcal{R})}$ as $\text{irc}_{\Phi(\mathcal{R})}$ limited to the basic terms of $\mathcal{R}$. We can start from the $\subseteq$ inclusion, by proving that for any rewrite sequence $t \to^n_{\mathcal{R}} u$ with $t \in \mathcal{T}_{\mathcal{B}}(\Sigma)$, there exists an equally long innermost rewrite sequence in $\Phi(\mathcal{R})$ starting with the same basic term t. In order to prove this statement we need to define a new type of rewrite relation. The reason is that certain full rewrite sequences in the original TRS could not be replicated by using the encoded rules in the same order, while also maintaining an innermost strategy.

Example 21. *Consider the TRS $\mathcal{W}$*

$$\alpha_1 : \mathsf{g} \to \mathsf{f}(\mathsf{a}, \underline{\mathsf{a}}) \qquad \alpha_2 : \mathsf{a} \to \underline{\mathsf{a}} \qquad \alpha_3 : \mathsf{f}(x, \underline{\mathsf{a}}) \to \mathsf{f}(x, \underline{\mathsf{a}})$$

and the encoded rules in $\mathcal{W}'$:

$$\alpha'_1 : \mathsf{g} \to \mathsf{f}(\mathsf{a}, \mathsf{i}(l_{\mathsf{a}})) \qquad \alpha'_2 : \mathsf{a} \to \mathsf{i}(l_{\mathsf{a}}) \qquad \alpha'_3 : \mathsf{f}(x, l_{\mathsf{a}}) \to \mathsf{f}(x, \mathsf{i}(l_{\mathsf{a}}))$$

Consider the following rewrite sequence

$$\mathsf{g} \quad \to_{\alpha_1} \quad \mathsf{f}(\mathsf{a}, \mathsf{a}) \quad \to_{\alpha_3} \quad \mathsf{f}(\mathsf{a}, \mathsf{a}) \quad \to_{\alpha_3} \cdots$$

This rewrite sequence cannot be constructed in $\Phi(\mathcal{W})$ and be innermost. The term a at position 1 of $\mathsf{f}(\mathsf{a}, \mathsf{a})$ is never encoded and thus not in normal form. So when trying to reproduce this rewrite sequence in $\Phi(\mathcal{W})$, it results in $\mathsf{h} \to_{\alpha'_1} \mathsf{f}(\mathsf{a}, \mathsf{i}(l_{\mathsf{a}}))$. At this point, there are not many options for innermost rewrite steps and the next 1-cost rule to be applied can be α'_2 and not α'_3 as in the original sequence.

This shows that even if the encoding, given some rewrite sequence as input, can produce an innermost rewrite sequence of the same length and with the same starting term, the order of 1-cost rule applications is not the same. Obviously, the end term is of no interest, but we need to ensure that such a rewrite sequence can be algorithmically produced.

One idea was to extend lemma 8 from [7] to TRSs with 0-cost rules, which states that for a regular TRS $\mathcal{R}$ the following holds: $t \overset{\text{ndg}}{\to}^n_{\mathcal{R}} u \Rightarrow t \overset{i}{\to}^n_{\mathcal{R}} v$. Since the set of regular TRSs is a subset of all TRSs with 0-cost rules, we can not take this statement for granted. Assuming we can prove it, we could easily show that an ndg rewrite sequence in the encoded TRS of the same length exists, while also maintaining the rule application order of the rewrite sequence in the original TRS. However, a counter-example was discovered using a non-terminating rewrite sequence.

Consider the following TRS with a constructor symbol 0 with arity 0, not present in the rules

$$\gamma_1 : \mathsf{f}(x) \to \mathsf{f}(\mathsf{a}) \quad \gamma_2 : \mathsf{a} \xrightarrow{0} \mathsf{a}$$

The rewrite sequence $\mathsf{f}(0) \to_{\gamma_1} \mathsf{f}(\mathsf{a}) \to_{\gamma_1} \mathsf{f}(\mathsf{a}) \to_{\gamma_1} \cdots$ is clearly of infinite length, but also ndg, as no duplication or the matching of nested redexes occurs. Meanwhile, the longest innermost rewrite sequence $\mathsf{f}(0) \to_{\gamma_1} \mathsf{f}(\mathsf{a}) \xrightarrow{0}_{\gamma_2} \mathsf{f}(\mathsf{a}) \xrightarrow{0}_{\gamma_2} \cdots$ has a length of one. Excluding non-terminating rewrite sequences, still left a difficult to prove statement even though no counter-examples were found.

The approach that was taken, is to shift the order of rule applications in the original sequence in such a way as to maintain its length and take that as the input for an algorithm that constructs the rewrite sequence in the encoded TRS. The input is such that the algorithm does not need to create a different order of rule applications. This shifting of rule application is not done via an algorithm, but the existence of these rewrite sequences is defined in much the same way as in lemma 8 from [7]. Basically, we want to show that for every full rewrite sequence of length n and starting term t, there is, what we call, an *optional innermost* rewrite sequence with the same length and starting term. The new rewrite relation is based on tagging, which works similarly to the calculation of non-ndg locations but cuts the over-approximation.

Definition 22 (Optional innermost)**.** *Given a rewrite sequence $\nabla = t_0 \to^n t_n$, a rewrite step in that sequence is **optional innermost*** (oi) *denoted $t_i \xrightarrow{\text{oi}}_\pi t_{i+1}$, if*

$$\textit{for all } \tau \in \text{pos}(t_i) \textit{ with } \tau > \pi \textit{ we have either } t_i|_\tau \textit{ in normal form or } \tau \in \text{tag}_\nabla(i)\ .$$

In order to better illustrate where this rewrite relation stands compared to others, we have the following order: $\xrightarrow{i} \subseteq \xrightarrow{\text{oi}} \subseteq \to$. For the first inclusion, we have that any innermost redex has no proper subterms not in normal form, therefore no position needs to meet the specified requirement. The second one is trivial. We can now move on to the tagging.

Definition 23 (tag_∇)**.**

1. *Given a finite rewrite sequence $\nabla = t_0 \to_{\alpha_1,\pi_1} t_1 \to \cdots \to_{\alpha_n,\pi_n} t_n$ in a TRS $\mathcal{R}$ we define $\text{tag}_\nabla(i) \subseteq \text{pos}(t_i)$ as the smallest set such that:*

   - $\{\tau \mid \tau = \pi_i.\upsilon, \quad (\alpha_i\ , \quad \mathtt{R}, w) \in \mathcal{X}'_\mathcal{R}, w \leq \upsilon\} \subseteq \text{tag}_\nabla(i)$
   - $\{\tau \mid \tau = \pi_{i+1}.\upsilon, (\alpha_{i+1}, \mathtt{L}, w) \in \mathcal{X}'_\mathcal{R}, w \leq \upsilon\} \subseteq \text{tag}_\nabla(i)$
   - $\left\{ \tau \;\middle|\; \begin{array}{l} \alpha_{i+1} = \ell_{i+1} \to r_{i+1}, \\ w_L \in pos(\ell_{i+1}), \quad w_R \in pos(r_{i+1}), \\ x = \ell_{i+1}|_{w_L} = r_{i+1}|_{w_R} \in \mathcal{V}, \\ \pi_{i+1}.w_L.\tau_x = \tau, \quad \pi_{i+1}.w_R.\tau_x = \gamma \end{array} \right\} \subseteq \text{tag}_\nabla(i),\ \textit{if } \gamma \in \text{tag}_\nabla(i+1)$
   - $\{\tau \mid \tau \not\leq \pi_{i+1} \qquad\qquad\qquad\qquad\qquad\quad \} \subseteq \text{tag}_\nabla(i),\ \textit{if } \tau \in \text{tag}_\nabla(i+1)$

2. *Given an infinite rewrite sequence $\nabla = t_0 \to_{\alpha_1,\pi_1} t_1 \to \cdots$, let ∇^n be the first n steps of ∇. Then*

$$\text{tag}_\nabla(i) = \bigcup_{k=i}^{\infty} \text{tag}_{\nabla^k}(i).$$

Going back to the rewrite sequence in Ex. 21 which showed we can not maintain rule application order in $\Phi(\mathcal{W})$, we get the following tagged rewrite sequence. Tagged positions are underlined.

$$\mathsf{g} \quad \to_{\alpha_1} \quad \mathsf{f}(\mathsf{a}, \underline{\mathsf{a}}) \quad \to_{\alpha_3} \quad \mathsf{f}(\mathsf{a}, \underline{\mathsf{a}}) \quad \to_{\alpha_3} \cdots$$

Keep in mind that the locations in the TRS considered by tag_∇ are only the base non-ndg locations. There is however an infinite oi rewrite sequence, which in our case is also innermost.

$$\mathsf{g} \quad \to_{\alpha_1} \quad \mathsf{f}(\mathsf{a}, \mathsf{a}) \quad \to_{\alpha_2} \quad \mathsf{f}(\mathsf{a}, \mathsf{a}) \quad \to_{\alpha_2} \cdots$$

The advantage of this type of tagging in a rewrite sequence is that it does not over-approximate like the rest of our analysis. If for example a nested redex is matched, then that subterm is tagged and the tagging is propagated backwards in the sequence, i.e. the subterms that flow into these tagged positions are tagged too. The over-approximation of the encoding ensures that in simulations of oi rewrite sequences, these tagged positions are encapsulated by the specially defined symbol i or are in normal form. We make use of this in the definition of the algorithm, which constructs the rewrite sequence in the encoded TRS, and therefore also in the proof.

Lemma 24. *For some TRS $\mathcal{R}$, if $t \to_{\mathcal{R}}^{n} v$ then $t \overset{\mathsf{oi}}{\to}_{\mathcal{R}}^{n} u$ for some term u.*

We claim lemma 24 without proof. Before we get to the algorithm, which constructs the rewrite sequences in the encoded TRS, consider the following rewrite sequence in TRS $\mathcal{P}$ from Ex. 19:

$$\mathsf{h} \overset{\mathsf{oi}}{\to}_{\alpha_1,\varepsilon} \mathsf{a}(\mathsf{f}(\mathsf{c})) \overset{\mathsf{oi}}{\to}_{\alpha_4,\varepsilon} \mathsf{b}(\mathsf{f}(\mathsf{c}),\mathsf{f}(\mathsf{c})) \overset{\mathsf{oi}}{\to}_{\alpha_3,1} \mathsf{b}(\mathsf{f}(\mathsf{c}),\mathsf{f}(\mathsf{c})) \overset{\mathsf{oi}}{\to}_{\alpha_5,1.1} \mathsf{b}(\mathsf{f}(\mathsf{0}),\mathsf{f}(\mathsf{c}))$$

Since the $\mathrm{irc}'_{\Phi(\mathcal{P})}$ only considers basic terms over the signature of $\mathcal{P}$, we can choose h as the starting term in the simulated rewrite sequence ∇'. We can now also apply the same reduction, i.e. apply α'_1 at ε to give us:

$$\nabla' = \mathsf{f}(\mathsf{s}(\mathsf{0})) \quad \overset{\mathsf{i}}{\to}_{\alpha'_1,\varepsilon} \quad \mathsf{a}(\mathsf{i}(l_{\mathsf{f}}(l_{\mathsf{c}})))$$

We would like to continue applying the encoded version of the rules in ∇ at the appropriate positions, but as we can see now, doing so would result in a rewrite step that is not innermost. Thus we have to remove the i-symbol at position 1. Only then can we append the next rule application to the rewrite sequence.

$$\begin{array}{llll} \nabla' = \mathsf{f}(\mathsf{s}(\mathsf{0})) & \overset{\mathsf{i}}{\to}_{\alpha'_1,\varepsilon} & \mathsf{a}(\mathsf{i}(l_{\mathsf{f}}(l_{\mathsf{c}}))) & \overset{\mathsf{i}}{\to}^{0}_{\beta_1,1} \quad \mathsf{a}(l_{\mathsf{f}}(l_{\mathsf{c}})) \\ & \overset{\mathsf{i}}{\to}_{\alpha'_4,\varepsilon} & \mathsf{b}(\mathsf{i}(l_{\mathsf{f}}(l_{\mathsf{c}})),\dots) & \end{array}$$

Since the subterms at positions 1 and 2 are duplicates, and no reductions are performed at position 2, we can ignore them to improve readability. The next rule application is supposed to be α'_3 at position 1. However, in the current term, the root symbol at that position is an i instead of the expected f. Therefore, when deciding the position of the redex, one must ignore the i-symbols. We call that a translation of the position and is formally defined later. For now, we know that the translated position is 1.1.

However, the root symbol at that position is a constructor and thus cannot be reduced. More specifically, it is a constructor, which can be restored to a defined symbol that it encodes. This is done by the i-symbol directly above that position. After it is restored, the term can be reduced via α'_3 at the translated position 1.1.

$$\begin{array}{llll} \nabla' = \mathsf{f}(\mathsf{s}(\mathsf{0})) & \overset{\mathsf{i}}{\to}_{\alpha'_1,\varepsilon} & \mathsf{a}(\mathsf{i}(l_{\mathsf{f}}(l_{\mathsf{c}}))) & \overset{\mathsf{i}}{\to}^{0}_{\beta_1,1} \quad \mathsf{a}(l_{\mathsf{f}}(l_{\mathsf{c}})) \\ & \overset{\mathsf{i}}{\to}_{\alpha'_4,\varepsilon} & \mathsf{b}(\mathsf{i}(l_{\mathsf{f}}(l_{\mathsf{c}})),\dots) & \overset{\mathsf{i}}{\to}^{0}_{\beta_2,1} \quad \mathsf{b}(\mathsf{i}(\mathsf{f}(l_{\mathsf{c}})),\dots) \\ & \overset{\mathsf{i}}{\to}_{\alpha'_3,1.1} & \mathsf{b}(\mathsf{i}^2(l_{\mathsf{f}}(l_{\mathsf{c}})),\dots) & \end{array}$$

The next step in ∇ reduces the nested c to $\mathsf{0}$. After translating the position to 1.1.1.1, we see that there is no i-symbol directly above it to restore the c-symbol. So an i-symbol has to be propagated from above. A function can return the position of the next closest non-parallel i-symbol above the translated position. If the defined symbol can still not be restored, then repeat the above steps until it can be restored. We can see that the position of that next i-symbol is 1.1. After the propagation, the translation of the original position has changed and thus has to be reevaluated. For the purpose of conciseness let $\pi = 1.1.1.1.1$.

$$\begin{array}{l} \nabla' = \\ \mathsf{f}(\mathsf{s}(\mathsf{0})) \end{array} \begin{array}{llllll} \overset{\mathsf{i}}{\to}_{\alpha'_1,\varepsilon} & \mathsf{a}(\mathsf{i}(l_{\mathsf{f}}(l_{\mathsf{c}}))) & \overset{\mathsf{i}}{\to}^{0}_{\beta_1,1} & \mathsf{a}(l_{\mathsf{f}}(l_{\mathsf{c}})) & & \\ \overset{\mathsf{i}}{\to}_{\alpha'_4,\varepsilon} & \mathsf{b}(\mathsf{i}(l_{\mathsf{f}}(l_{\mathsf{c}})),\dots) & \overset{\mathsf{i}}{\to}^{0}_{\beta_2,1} & \mathsf{b}(\mathsf{i}(\mathsf{f}(l_{\mathsf{c}})),\dots) & & \\ \overset{\mathsf{i}}{\to}_{\alpha'_3,1.1} & \mathsf{b}(\mathsf{i}^2(l_{\mathsf{f}}(l_{\mathsf{c}})),\dots) & \overset{\mathsf{i}}{\to}^{0}_{\beta_3,1.1} & \mathsf{b}(\mathsf{i}^2(l_{\mathsf{f}}(\mathsf{i}(l_{\mathsf{c}}))),\dots) & \overset{\mathsf{i}}{\to}^{0}_{\beta_4,\pi} & \mathsf{b}(\mathsf{i}^2(l_{\mathsf{f}}(\mathsf{i}(\mathsf{c}))),\dots) \\ \overset{\mathsf{i}}{\to}_{\alpha_5,\pi} & \mathsf{b}(\mathsf{i}^2(l_{\mathsf{f}}(\mathsf{i}(\mathsf{0}))),\dots) & & & & \end{array}$$

As we can see ∇' has the same length as ∇, so we can conclude that the input sequence can be simulated in the encoded TRS. This construction can be defined algorithmically. However, since we encountered two tangential problems, while constructing ∇', namely the *translation of positions* and the *calculation of the closest* i*-symbol above a position*, next we define them formally.

Definition 25 (Position translation)**.** *Given some term t over the signature of an encoded TRS and a position π we define the following function:*

$$\mathrm{tr}(t,\pi) = \begin{cases} 1.\mathrm{tr}(t|_1,\pi) & \textit{if} \quad \mathrm{root}(t) = \mathsf{i} \\ \pi_1.\mathrm{tr}(t|_{\pi_1},\pi_2) & \textit{else if} \quad \pi = \pi_1.\pi_2 \textit{ such that } \pi_1 \in \mathbb{N} \\ \varepsilon & \textit{else} \end{cases}$$

For example, consider the term $\mathsf{b}(\mathsf{i}(l_\mathsf{f}(l_\mathsf{c})),\dots)$ from ∇', then

$$\mathrm{tr}(\mathsf{b}(\mathsf{i}(l_\mathsf{f}(l_\mathsf{c})),\dots),\ 1) \ =\ 1.\mathrm{tr}(\mathsf{i}(l_\mathsf{f}(l_\mathsf{c})),\ \varepsilon) \ =\ 1.1.\mathrm{tr}(l_\mathsf{f}(l_\mathsf{c}),\ \varepsilon) \ =\ 1.1.$$

Definition 26 (Next i above)**.** *Given some term t over the signature of an encoded TRS and a position π we define the following function:*

$$\mathrm{n\_i}(t,\pi) = \begin{cases} \pi & \textit{if} \quad \mathrm{root}(t|_\pi) = \mathsf{i} \\ \mathrm{n\_i}(t,\pi_1) & \textit{else if} \quad \pi = \pi_1.\pi_2 \textit{ such that } \pi_2 \in \mathbb{N} \\ \bot & \textit{else} \end{cases}$$

Consider the term $t = \mathsf{b}(\mathsf{i}^2(l_\mathsf{f}(l_\mathsf{c})),\dots)$ from ∇' which was when we needed to find the closest i-symbol above the position 1.1.1.1. Then we have

$$\begin{aligned} \mathrm{n\_i}(t,\ 1.1.1.1) &= \mathrm{n\_i}(t,\ 1.1.1) && \text{, since } l_\mathsf{c} \neq \mathsf{i} \\ &= \mathrm{n\_i}(t,\ 1.1) && \text{, since } l_\mathsf{f} \neq \mathsf{i} \\ &= 1.1 && \text{, since } \mathrm{root}(t|_{1.1}) = \mathsf{i} \end{aligned}$$

The algorithm is designed in such a way as to never call n_i on an input that would return $\bot$, but in order to give a complete definition it is included.

Algorithm 27 (`CreateEncodedSequence`)**.**

1: **procedure** CreateEncodedSequence($\mathcal{R}$, $t_0 \overset{\text{oi}}{\to}_{\alpha_1,\pi_1} \cdots \overset{\text{oi}}{\to}_{\alpha_n,\pi_n} t_n$)
2: $s_{0,0} \leftarrow t_0$
3: $k \leftarrow 0$

4: **while** $k < n$
5: $j \leftarrow 0$
6: $w \leftarrow \mathrm{tr}(s_{k,j}, \pi_{k+1})$

7: **while** $s_{k,j}|_w$ is not innermost
8: Let $w' \geq w$ such that $\mathrm{root}(s_{k,j}|_{w'}) = \mathsf{i}$ and $s_{k,j}|_{w'}$ is innermost.
9: $\beta_{k,j+1} \leftarrow \mathtt{OMIT}$
10: $\tau_{k,j+1} \leftarrow w'$
11: $\sigma \leftarrow \mathrm{MGU}\,(\mathtt{OMIT}^L, s_{k,j}|_{\tau_{k,j+1}})$
12: $s_{k,j+1} \leftarrow s_{k,j}[\mathtt{OMIT}^R\sigma]_{\tau_{k,j+1}}$
13: $j \leftarrow j + 1$
14: **end while**

15: **if** $\pi_{k+1} \neq w$
16: Let $w_1.w_2 \leftarrow w$ such that $w_2 \in \mathbb{N}$
17: **while** $\mathrm{root}(s_{k,j}|_{w_1}) \neq \mathsf{i}$
18: $q \leftarrow \mathrm{n\_i}(s_{k,j}|_{w_1})$
19: $f \leftarrow \mathrm{root}(s_{k,j}|_{q.1})$
20: $\beta_{k,j+1} \leftarrow \mathtt{PROP}_f$
21: $\tau_{k,j+1} \leftarrow q$
22: $\sigma \leftarrow \mathrm{MGU}\,(\mathtt{PROP}^L, s_{k,j}|_{\tau_{k,j+1}})$
23: $s_{k,j+1} \leftarrow s_{k,j}[\mathtt{PROP}^R\sigma]_{\tau_{k,j+1}}$
24: $w \leftarrow \mathrm{tr}(s_{k,j+1}, \pi_{k+1})$
25: Let $w_1.w_2 \leftarrow w$ such that $w_2 \in \mathbb{N}$
26: $j \leftarrow j + 1$
27: **end while**

28: $f \leftarrow \mathrm{root}(s_{k,j}|_w)$
29: $\beta_{k,j+1} \leftarrow \mathtt{EXE}_f$
30: $\tau_{k,j+1} \leftarrow w_1$
31: $\sigma \leftarrow \mathrm{MGU}\,(\mathtt{EXE}^L, s_{k,j}|_{\tau_{k,j+1}})$
32: $s_{k,j+1} \leftarrow s_{k,j}[\mathtt{EXE}^R\sigma]_{\tau_{k,j+1}}$
33: **end if**

34: $\alpha'_{k+1} \leftarrow \psi(\alpha_{k+1})$
35: $\pi'_{k+1} \leftarrow w$
36: $\sigma \leftarrow \mathrm{MGU}\,(\varphi(\ell_{k+1}), s_{k,j}|_{\tau_{k,j+1}})$
37: $s_{k,j+1} \leftarrow s_{k,j}[\varphi(r_{k+1})\sigma]_{\pi_{k+1}}$
38: $k \leftarrow k + 1$
39: **end while**

40: **return** $s_{0,0} \overset{0}{\to}_{\beta_{0,1},\tau_{0,1}} \cdots \overset{0}{\to}_{\beta_{0,m_1},\tau_{0,m_1}} s_{0,m_1} \to_{\alpha'_1,\pi'_1} s_{1,0} \to \cdots \to_{\alpha'_n,\pi'_n} s_{n,0}$
41: **end procedure**

We can now go into further detail and better illustrate how the algorithm works. After initialization, i.e. assigning the same starting term as the one from the input sequence and setting the counter k to 0 for the big while-loop spanning lines 4 to 39. Each iteration of this loop constructs the next rewrite step of the output, which may also include some rewrite steps using the 0-cost rules, but always ends by applying a single 1-cost rule. In other words, the loop iterates over each rewrite step of the input rewrite sequence and simulates it in the encoded TRS.

Next, we take a deeper look at each iteration. Lines 5 and 6 initialize a second counter j, which counts the 0-cost rewrite steps, and a position w, set to the translation of π_{k+1} in the current term. Position π_{k+1} refers to the position of the redex at t_k in the input rewrite sequence.

The while-loop from lines 7 to 14 removes all i-symbols, which are below the redex, using the `OMIT` rule. This part of the procedure ensures that subsequent rewrite steps are innermost. We will show later why considering only the i-symbols is sufficient. In the algorithm, β and τ are reserved for the details of the 0-cost rewriting, namely the rule and position respectively.

Line 8 chooses the position w' of a subterm with root symbol i, which is also innermost. This can be performed automatically via a depth-first search. The next lines assign the proper values to evaluate at w' and update the value of j.

After the redex at w has been made innermost, the procedure checks if w equals the position π_{k+1} from the input rewrite sequence. If that is the case, then lines 34 to 38 directly apply the encoded version of the rule used in the rewrite step, that the algorithm is currently simulating. Otherwise, as we will show later, the position is below some i-symbol and is also not a redex yet. Therefore, the algorithm propagates an i-symbol directly above the subterm at w in lines 16 to 27 and restores its encoded symbol to the original defined symbol in lines 28 to 32.

Line 16 splits w in a way as to give us the position above it, namely w_1. Then the while-loop at lines 17 to 27 checks after each iteration if the symbol at that position is an i-symbol. The first two lines of the loop initialize a position q to the position of the closest i-symbol above w_1, and then assigns f the symbol directly below the discovered i-symbol. More precisely that would be the root symbol at $q.1$. The next lines define the next 0-cost rewrite step analogously to the one seen in the previous while-loop. Since the propagation of i-symbols changes the structure of the term, the algorithm must update the value of w.

When the algorithm reaches line 28 after exiting the previous while-loop, the exact conditions to apply `EXE` at w_1 are met and the defined symbol is restored at w. The goal of the if-branch from lines 15 to 33 was to create a redex at position w, where none was there prior to it. The final 1-cost rule application is therefore shared by both cases. Line 38 increments k in preparation for the next rewrite step. After n many steps, where n is the length of the input rewrite sequence, the big while-loop terminates and the procedure returns the innermost rewrite sequence in the encoded $\Phi(\mathcal{R})$. As we move towards the proof using the described algorithm, we define a function to decode a term over the signature of some encoded TRS $\Phi(\mathcal{R})$.

Definition 28 (Term decoding)**.** *Let Σ and Σ' be the signatures of $\mathcal{R}$ and $\Phi(\mathcal{R})$ respectively. Let $t \in \mathcal{T}(\Sigma', \mathcal{V})$ with $t = f(t_0, \ldots, t_n)$. We define the function*

$$\mathrm{dec}(t) = \begin{cases} \mathrm{dec}(t|_1) & \text{if} \quad f = \mathsf{i} \\ g(\mathrm{dec}(t_1), \ldots, \mathrm{dec}(t_n)) & \text{else if} \quad f = l_g \in \Sigma' \backslash \Sigma \\ f(\mathrm{dec}(t_1), \ldots, \mathrm{dec}(t_n)) & \text{else} \end{cases}$$

As a brief example consider the decoding of $\mathsf{a}(\mathsf{i}(l_\mathsf{f}(l_\mathsf{c})))$ taken from the signature of $\Phi(\mathcal{W})$ from Ex. 21:

$$\mathrm{dec}(\mathsf{a}(\mathsf{i}(l_\mathsf{f}(l_\mathsf{c})))) = \mathsf{a}(\mathrm{dec}(\mathsf{i}(l_\mathsf{f}(l_\mathsf{c})))) = \mathsf{a}(\mathrm{dec}(l_\mathsf{f}(l_\mathsf{c}))) = \mathsf{a}(\mathsf{f}(\mathrm{dec}(l_\mathsf{c}))) = \mathsf{a}(\mathsf{f}(\mathsf{c}))$$

Lemma 29 ($\mathrm{rc}_\mathcal{R}(n) \leq \mathrm{irc}'_{\Phi(\mathcal{R})}(n)$)**.** *Let $\mathcal{R}$ be a TRS, Σ the signature of $\mathcal{R}$, and $\Phi(\mathcal{R})$ its encoding. Let $t \in \mathcal{T}_\mathcal{B}(\Sigma)$ be some basic term.*

$$t \to^n_\mathcal{R} v \quad \Rightarrow \quad t \xrightarrow{i}{}^n_{\Phi(\mathcal{R})} u$$

Proof. We apply lemma 24. to $t \to^n_\mathcal{R} v$ and obtain $\nabla = t \xrightarrow{\mathsf{oi}}{}^n_\mathcal{R} v'$ for some term v'. The procedure CreateEncodedSequence$(\mathcal{R}, \nabla)$ returns a rewrite sequence ∇' in $\Phi(\mathcal{R}) = \mathcal{R}' \cup \mathcal{S}$, which has n-many 1-cost rewrite steps and finite 0-cost rewrite steps between them. We know the following statements hold:

(1): Let $\nabla = t_0 \to_\mathcal{R} \cdots \to_\mathcal{R} t_n$ and $\nabla' = t'_0 \to_{\mathcal{R}'} v_0 \xrightarrow{0}_\mathcal{S} \cdots \xrightarrow{0}_\mathcal{S} t'_1 \to_{\mathcal{R}'} \cdots \to_{\mathcal{R}'} t'_n$. Then for all $0 \leq k \leq n$, $t_k = \mathrm{dec}(t'_k)$ and for all $\tau \in \mathrm{tag}_\nabla(k)$, $t'_k|_\tau$ is in normal form or there exists $\tau' \leq \mathrm{tr}(t'_k, \tau)$, with $\mathrm{root}(t'_k|_{\tau'}) = \mathsf{i}$.

Statement (1) says that for any term t_k in a rewrite sequence ∇ an equivalent after decoding t'_k is reachable in $\Phi(\mathcal{R})$ with subterms at translated tagged positions of t_k either in normal form or encapsulated by i.

(2): Let $\nabla' = q_0 \to_{\alpha_1} \cdots \to_{\alpha_n} q_j$. Then for all $0 \leq k \leq j$, $\pi \in \text{pos}(q_k)$ with $\text{root}(q_k|_\pi) = \mathsf{i}$, it holds for all $\tau \geq \pi$, $\text{root}(q_k|_\tau) = \mathsf{i}$, or $\text{root}(q_k|_\tau) \notin \Sigma_d$, or $(\text{root}(q_k|_\tau) \in \Sigma_d$ and $\tau = \pi$ with the rule $\alpha_k = \texttt{EXE}_f)$

Statement (2) says that any subterm found below an i-symbol in an innermost rewrite sequence in an encoded TRS has as root symbol either another i, or is not defined, and in the third case where it is defined, that the previous rule applied was an `EXE` rule.

(3): Let $q \in \mathcal{T}(\Sigma')$ and $\pi \in \text{pos}(q)$, where Σ' is the signature of $\Phi(\mathcal{R})$. Then either for all $\tau \geq \pi, \text{root}(q|_\tau) \neq \mathsf{i}$ or there exists a $\tau \geq \pi, \text{root}(q|_\tau) = \mathsf{i}$ with $q|_\tau$ an innermost redex.

Statement (3) says that for any term q over the signature of an encoded TRS and a position π in t, there either exists no positions with an i-symbol below π, or there exists such a position, which is also an innermost redex.

(4): Let $q \in \mathcal{T}(\Sigma')$ and $\pi \in \text{pos}(q)$, where Σ' is the signature of $\Phi(\mathcal{R})$. Then either $\text{n\_i}(q, \pi) = \bot$ or $\text{n\_i}(q, \pi) = \tau$ with $q|_\tau$ an innermost redex.

Statement (4) says that the result of $\text{n\_i}(q, \pi)$ for some t over the signature of an encoded TRS and a position π in t, is either $\bot$ or the position of an innermost redex.

In order to prove lemma 29, we show the following statements to hold:

I. ∇' exists in $\Phi(\mathcal{R})$

The starting term of ∇ is defined as basic and the procedure copies it for the starting term of its output. For the first iteration of the big while-loop, we know that $w = \pi_1 = \epsilon$ and that $s_{0,0}$ is already innermost. The procedure then goes directly to line 34 and applies the encoded $\psi(\alpha_1)$. We know that $\varphi(\ell_1) = \ell_1$ since the only time φ changes the left-hand side of a rule is when there is a nested defined symbol. This is not the case here, since the starting term is basic and therefore cannot be matched to such a left-hand side of any rule. Afterwards, k is incremented and the loop goes into the next iteration.

The loop has reached term $s_{1,0}$. The reasoning behind every rewrite step going forward is analogous, therefore we show how the procedure operates for terms $s_{k,0}$, where $1 \leq k < n$.

After the first rule application, it is possible that the next produced term no longer equals its counterpart in the input rewrite sequence. That would be due to encoding adding i-symbols at the over-approximated non-ndg locations of the TRS. That is why the procedure has to translate the position of the next redex from the input.

Regardless of the translation, we want the algorithm to produce an innermost rewrite sequence. Since the corresponding rewrite step in the input is oi, we can conclude that the positions of all subterms below π_{k+1} not in normal form are in $\text{tag}_\nabla(k)$. It then follows from (1-2) that removing all i-symbols at positions below the translated w position ensures that the subterm at it is an innermost redex or in normal form. It follows from (3) that every `OMIT` rewrite step in the produced rewrite sequence is also innermost.

The next part of the procedure is to check if the translated position is equal to π_{k+1}. This case distinction is needed for the situations when the redex is determined to flow into a non-ndg location, i.e. the subterm at w is encapsulated by an i-symbol.

Case 1. ($\pi_{k+1} \neq w$) From (2) we also know that the root symbol at w is a constructor, which by the Def. 25 can be restored to some defined symbol, i.e. it is not an i-symbol. Since it is not necessary that the root symbol at w_1, i.e. the position directly above w, is an i-symbol, the procedure has to propagate one to it from the closest position above containing an i-symbol, which is conveniently returned from the function n_i.

This is done by the while-loop from lines 17 to 27, which terminates once an i-symbol is directly above w. Each iteration calculates the new closest non-parallel position with root symbol i and propagates at it. From (4) and since all other i-symbols at positions below w have been removed it follows that all these `PROP` rewrite steps are innermost. When this while-loop terminates, the `EXE` rewrite step restores the defined symbol at the now updated w.

Case 2. ($\pi_{k+1} = w$) In this case we know that the subterm at w is already an innermost redex. For subterms that were not in normal form, the while-loop from lines 7 to 14 removed all nested i-symbols and from (2) we know all that is left at these positions are constructors.

Convergence of both cases. At this point we have some term $s_{k,j}$, and an innermost redex at position w. We now have to show that the redex can be matched to ℓ_{k+1}. It follows from the

procedure so far that $\mathrm{dec}(s_{k,j}) = t_k$. If there are nested defined symbols in ℓ_{k+1}, then these are encoded by Φ to their constructor versions. These locations always tag positions in t_k and from (1) we know that the constructor version of said symbol is also present at the same position in $s_{k,j}$. Therefore, the encoded rule $\psi(\alpha_{k+1})$ can be applied w and we are done. If there are no nested defined symbols in ℓ_{k+1}, then the matching is trivial and we are done.

2. ∇' is an innermost rewrite sequence

Most of this has already been shown in the part above, namely statements $(3-4)$ covering the removal and propagation of i-symbols. And since the `PROP` rewrite steps are innermost, then it follows the `EXE` step is also innermost, since it evaluates at a position where an i-symbol was lastly propagated. Consequently, the last 1-cost rewrite step per iteration is also innermost.

3. ∇' is has a length of n

The length of rewrite sequences is only affected by 1-cost rewrite steps. The big while-loop in the procedure produces a single 1-cost rewrite step per iteration and it iterates n many times, where n is the length of the input rewrite sequence.

4. The procedure CreateEncodedSequence always terminates

There are three while-loops in the procedure. The big while-loop that contains the other two, terminates trivially since the parameter k is incremented at the end of each iteration and is not changed elsewhere.

The while-loop from lines 7 to 14 terminates once all i-symbols at positions below position w are removed. Since no other i-symbols are introduced in the process and there are finitely many of them at the start, we can conclude this loop also always terminates.

The last while-loop to consider is from lines 17 to 27. Before entering this loop it is ensured that there is a position above w which contains a subterm with root symbol i. The process of propagation does not remove any i-symbols in the process and therefore will eventually terminate.

The two other function calls in the procedure, namely tr and n_i, do calculate recursively, but they obviously always terminate.

The proof of these statements suffices to prove lemma 29.

□

Lemma 30. *Let $\mathcal{R}$ be a TRS and $\Phi(\mathcal{R}) = \mathcal{R}' \cup \mathcal{S}$ its encoding.*

$$t \xrightarrow{i}_{\Phi(\mathcal{R})} v \quad \Rightarrow \quad dec(t) \to_{\mathcal{R}} dec(v)$$

Proof. Represent $t \xrightarrow{i}_{\Phi(\mathcal{R})} v$ as

$$t = t_0 \xrightarrow{i}_{\mathcal{S}} \cdots \xrightarrow{i}_{\mathcal{S}} t_n \xrightarrow{i}_{\mathcal{R}'} v_0 \xrightarrow{i}_{\mathcal{S}} \cdots \xrightarrow{i}_{\mathcal{S}} v_m = v\,.$$

It holds that $\mathrm{dec}(t_j) = \mathrm{dec}(t_k)$, for all j, k. Since the decoder removes all i-symbols and restores the original defined symbol of it encounters its constructor version, no amount of rewriting with the rules in $\mathcal{S}$ can change the decoding of the term. Analogously, $\mathrm{dec}(v_j) = \mathrm{dec}(v_k)$, for all j, k. Therefore, proving $t_n \xrightarrow{i}_{\mathcal{R}'} v_0 \Rightarrow \mathrm{dec}(t_n) \to_{\mathcal{R}} \mathrm{dec}(v_0)$ also shows $t \xrightarrow{i}_{\Phi(\mathcal{R})} v \Rightarrow \mathrm{dec}(t) \to_{\mathcal{R}} \mathrm{dec}(v)$. Let $t_n \to_{\psi(\alpha),\pi} v_0$. We define τ via $\pi = \mathrm{tr}(t_n, \tau)$. Let $t_n = C[\varphi(\ell)\sigma]$. Define context $D = \mathrm{dec}(C)$ with $D|_\tau = \Box$. We have

$$\mathrm{dec}(t_n) = \mathrm{dec}(C[\varphi(\ell)\sigma]) = D[\mathrm{dec}(\varphi(\ell)\sigma)] = D[\ell\sigma'],$$

where $\sigma = \{x_0 \backslash q_0,\, x_1 \backslash q_1,\, \ldots,\, x_p \backslash q_p\}$ and $\sigma' = \{x_0 \backslash \mathrm{dec}(q_0),\, x_1 \backslash \mathrm{dec}(q_1),\, \ldots,\, x_p \backslash \mathrm{dec}(q_p)\}$. Therefore

$$\mathrm{dec}(v_0) = \mathrm{dec}(C[\varphi(r)\sigma]) = D[\mathrm{dec}(\varphi(r)\sigma)] = D[r\sigma'].$$

This gives us the rewrite step

$$\mathrm{dec}(t) = \mathrm{dec}(t_n) = D[\ell\sigma'] \to_{\alpha,\tau} D[r\sigma'] = \mathrm{dec}(v_0) = \mathrm{dec}(v)$$

and we are done.

□

Lemma 31 ($\mathrm{rc}_{\mathcal{R}}(n) \geq \mathrm{irc}'_{\Phi(\mathcal{R})}(n)$)**.** *Let $\mathcal{R}$ be a TRS with signature Σ and $\Phi(\mathcal{R})$ its encoding. Let $t \in \mathcal{T}(\Sigma)$ be some basic term.*

$$t \xrightarrow{i}{}^{n}_{\Phi(\mathcal{R})} v \quad \Rightarrow \quad t \to^{n}_{\mathcal{R}} \mathrm{dec}(v)$$

Proof. For $n = 0$ it is trivial. For $n > 0$ apply the induction hypothesis on the first $n - 1$ steps.

$$t \xrightarrow{i}{}^{n-1}_{\Phi(\mathcal{R})} u \xrightarrow{i}_{\Phi(\mathcal{R})} v \quad \Rightarrow \quad t \to^{n-1}_{\mathcal{R}} \mathrm{dec}(u)$$

We apply lemma 30 on the last $u \xrightarrow{i}_{\Phi(\mathcal{R})} v$ step to get $\mathrm{dec}(u) \to_{\mathcal{R}} \mathrm{dec}(v)$. We now have the sequence

$$t \to^{n}_{\mathcal{R}} \mathrm{dec}(v)$$

and we are done.

□

Theorem 32. *Let $\mathcal{R}$ be a TRS and $\Phi(\mathcal{R})$ its encoding. Then $\mathrm{rc}_{\mathcal{R}} = \mathrm{irc}'_{\Phi(\mathcal{R})}$.*

Proof. $\mathrm{rc}_{\mathcal{R}}(n) \leq \mathrm{irc}'_{\Phi(\mathcal{R})}(n)$ follows from lemma 29. The opposite direction $\mathrm{rc}_{\mathcal{R}}(n) \geq \mathrm{irc}'_{\Phi(\mathcal{R})}(n)$ follows from lemma 31.

In this chapter, we explained the concept of encoding a TRS and explored further its usefulness in a few examples. The encoding itself was defined and proven to have the desired property, namely its modified innermost runtime complexity equals the full runtime complexity of the original TRS. In the next chapter, we tackle the second main problem mentioned in the introduction, namely the inclusion of non-terminating 0-cost rules by the encoding. An expanded encoding, which addresses this issue is presented.

4. Expanded Encoding

The introduced encoding Φ is completely sufficient when it comes to theoretical work. For the calculation of an upper bound on rc of some TRS $\mathcal{R}$ all that the encoding shows is that given a rewrite sequence in $\mathcal{R}$, there exists in the infinite set of rewrite sequences in $\Phi(\mathcal{R})$ a rewrite sequence starting with the same basic term and of the same length. We created an algorithm to find these rewrite sequences, but the automatic complexity analysis can not account for the non-terminating rules added by Φ. As we stated in the introduction, limiting the addition of non-terminating rules is one of the two main problems. Now that one of them has been covered, we turn our attention to this: what alternatives are there to the addition of non-terminating rules, and how can we best limit their addition by the encoding? In this chapter, we theorize an improvement of the encoding, which addresses these questions.

We know so far that the propagation rules of any kind added by Φ are absolutely necessary. However, consider this simple TRS:

$$\alpha_1 : \mathsf{a} \to \mathsf{dbl}(\mathsf{b}(\mathsf{b}(0))) \quad \alpha_2 : \mathsf{dbl}(x) \to \mathsf{d}(x, x) \quad \alpha_3 : \mathsf{b}(x) \to 0$$

It is clear that $\psi(\alpha_1) = \mathsf{a} \to \mathsf{dbl}(\mathsf{i}(l_\mathsf{b}(l_\mathsf{b}(0))))$ and that the encoding will add the non-terminating $\mathtt{PROP}_\mathsf{b}$. But we can also see that, if the encoding was to add two i-symbols at that position in $\psi(\alpha_1)$ instead of just one, then each can serve either of the two nested redexes. This eliminates the need to add a non-terminating $\mathtt{PROP}_\mathsf{b}$ rule.

However, straightforward cases like this one are rare. Therefore, we must once again consider the flow of locations to ensure we do not add too few i-symbols, which could lead to a broken encoding. Consider the following example, one that may also have practical purposes.

Example 33 (Random trees)**.** *In this example, a tree data structure (represented by the symbols* tree *and* leaf*) is randomly constructed by TRS $\mathcal{R}_{tree}$. Rewrite sequences of the same length can produce vastly different trees, but the common thing behind them is they are all of at most depth $|t| - 1$ for a basic starting term t. Most importantly rule γ is there to prune the tree at any node that branches twice, i.e. both of its children are also trees.*

$$\begin{aligned}
&\alpha_1 : \mathsf{f}(\mathsf{s}(x)) \to \mathsf{tree}(\mathsf{s}(x), \mathsf{f}(x), \quad \mathsf{f}(x))\\
&\alpha_2 : \mathsf{f}(\mathsf{s}(x)) \to \mathsf{tree}(\mathsf{s}(x), \mathsf{leaf}(x), \mathsf{f}(x))\\
&\alpha_3 : \mathsf{f}(\mathsf{s}(x)) \to \mathsf{tree}(\mathsf{s}(x), \mathsf{f}(x), \quad \mathsf{leaf}(x))\\
&\alpha_4 : \mathsf{f}(\mathsf{s}(x)) \to \mathsf{leaf}(\mathsf{s}(x))\\
&\alpha_5 : \mathsf{f}(0) \quad\ \to \mathsf{leaf}(0)
\end{aligned}$$

$$\gamma : \mathsf{tree}(\mathsf{s}(x), \mathsf{tree}(y), \mathsf{tree}(z)) \to \mathsf{leaf}(0)$$

It is obvious that the resulting terms would nest multiple redexes with root symbol tree*. In fact, the larger the input is, the more nesting occurs. It is not predictable, since it depends on the input. Adding the terminating version of $\mathtt{PROP}_\mathsf{tree}$ and encapsulating the right-hand side of all of the α_1 to α_3 with a single* i*-symbol, will not be able to simulate all rewrite sequences, because after pruning at some position in the tree, $\mathcal{R}_{tree}$ will no longer be able to prune at a position above.*

Assume then that all left-hand sides of rules α_1 to α_3 were replaced exactly by the term $\mathsf{f}(\mathsf{s}(0))$. Then we are looking at a constant runtime complexity and a maximum nesting of redexes equal to two. Thus adding two i-symbols in front of the right-hand side of rules α_1 to α_3 and encoding the rest is sufficient to allow the encoding to add the terminating $\mathtt{PROP}_\mathsf{tree}$. Further replacing the left-hand sides of these rules with $\mathsf{f}(\mathsf{s}^2(0))$ instead, means the encoding has to add three i-symbols, and so on.

Therefore, we differentiate between locations, which lead to potentially infinite nesting of redexes, and locations, which have a maximum nesting depth. We chose to introduce the definition of the former first, which will also help with the definition of the latter when we exclude all these locations from the maximum depth analysis. When the expanded encoding adds the propagation rule for some symbol f, it will consider, if any location which potentially leads to infinite nesting, has the root symbol f. If there is such a location, then the non-terminating $\mathtt{PROP}_f$ is added, otherwise the terminating one is added.

It is clear that locations, that flow into themselves or into one of their sub-locations, have the potential of infinite nesting depth. In order to find if that is the case for a location λ, we can first define the set of location to which it flows via $\mathcal{Y}_{\mathcal{R}}^{\{\lambda\}}$. Since this set also includes λ itself, it has to be removed and afterward conduct a second forward data flow analysis with the resulting set, to determine if any of these locations flow back into λ. This can be considered the base case for all locations that flow into themselves. We need to further include locations with any sub-location that loops, and locations that are on the receiving end of a data flow with a location that loops.

Definition 34 ($\mathrm{INF}_{\mathcal{R}}$)**.** *For a TRS $\mathcal{R}$ let the* ***set of locations of potential infinite nesting depth*** $\mathrm{INF}_{\mathcal{R}}$ *be defined as the smallest set such that:*

- $\{\lambda \mid \mu \neq \lambda,\ \mu \in \mathrm{loc}(\lambda),\ \mu \in \mathcal{Y}_{\mathcal{R}}^{\{\lambda\}}\} \subseteq \mathrm{INF}_{\mathcal{R}}$ $(I1)$
- $\{\lambda \mid \mu \in \mathrm{loc}_{\mathcal{R}}(\lambda)\} \subseteq \mathrm{INF}_{\mathcal{R}},\ \textit{if } \mu \in \mathrm{INF}_{\mathcal{R}}$ $(I2)$
- $\{\lambda \mid \lambda \in \mathcal{Y}_{\mathcal{R}}^{\{\mu\}} \quad\} \subseteq \mathrm{INF}_{\mathcal{R}},\ \textit{if } \mu \in \mathrm{INF}_{\mathcal{R}}$ $(I3)$

The base case locations are included in $(I1)$. From them $(I2)$ includes all locations above these base case ones and $(I3)$ includes all locations, which either directly or at a sub-location receive a flow from an infinite looping location.

In Ex. 33 all right-hand sides of rules α_1 to α_3 contain a location with infinite nesting depth. That is because each of the subterms with root symbol f flows back into themselves since these are also their return locations. Resulting in an encoding that encapsulates the terms with root symbol tree with a single i-symbol, and non-terminating versions of $\mathtt{PROP}_{\mathsf{tree}}$ and $\mathtt{PROP}_{\mathsf{f}}$.

With this exclusion of locations with potentially infinite nesting depth, we can now focus on the locations that do not have this property and thus can be encoded with multiple i-symbols encapsulating them and with terminating PROP rules for their symbols. As is expected, we must track the flow of these locations so that the encoding adds the proper amount of i-symbols there too.

Definition 35 ($\mathrm{NST}_{\mathcal{R}}$)**.** *A* ***nest*** *is a tuple (λ, n), where λ is a location in some TRS $\mathcal{R}$ and n is its nesting depth.*
Let $\aleph_{\mathcal{R}}^0 := \{(\lambda,1) \mid \mathrm{root}(\mathcal{R}|_\lambda) \in \Sigma_d, \lambda \notin \mathrm{INF}_{\mathcal{R}}\} \cup \{(\lambda,0) \mid \mathrm{root}(\mathcal{R}|_\lambda) \notin \Sigma_d, \lambda \notin \mathrm{INF}_{\mathcal{R}}\}$. For $i \geq 0$:

1. *Define a function that will calculate the nesting depths properly after every iteration.*
   - $\mathrm{nest}_{\mathcal{R}}^i(\lambda) = \begin{cases} n & \textit{if} \quad (\lambda,n) \in \aleph_{\mathcal{R}}^i \ \textit{and} \\ & \quad \mathrm{loc}_{\mathcal{R}}(\lambda)\backslash\{\lambda\} = \varnothing \\ \max\{\mathrm{nest}_{\mathcal{R}}^i(\mu) \mid \mu \in \mathrm{loc}_{\mathcal{R}}(\lambda)\backslash\{\lambda\}\} & \textit{else if} \ \mathrm{root}(\mathcal{R}|_\lambda) \notin \Sigma_d \\ 1 + \max\{\mathrm{nest}_{\mathcal{R}}^i(\mu) \mid \mu \in \mathrm{loc}_{\mathcal{R}}(\lambda)\backslash\{\lambda\}\} & \textit{else} \end{cases}$
2. *Next, define the set of nests for a TRS $\mathcal{R}$. Let $\mathcal{N}_{\mathcal{R}}^i$ be the smallest set such that:*
   - $\{(\lambda,n) \mid n = \mathrm{nest}_{\mathcal{R}}^{i-1}(\lambda), \lambda \notin \mathrm{INF}_{\mathcal{R}}\} \subseteq \mathcal{N}_{\mathcal{R}}^i$
   - $\{(\mu,n) \mid \mu \in \mathcal{Y}_{\mathcal{R}}^{\{\lambda\}}, \lambda \notin \mathrm{INF}_{\mathcal{R}}\} \quad \subseteq \mathcal{N}_{\mathcal{R}}^i,\ \textit{if } (\lambda,n) \in \mathcal{N}_{\mathcal{R}}^i$
3. *The third step eliminates duplicates in the calculated set $\mathcal{N}_{\mathcal{R}}^i$, since more than one nest can flow to a location.*
   - $\aleph_{\mathcal{R}}^i := \{(\lambda,n) \mid n = \max\{m \mid (\lambda,m) \in \mathcal{N}_{\mathcal{R}}^i\}\}$

After finite many iterations, $\aleph_{\mathcal{R}}^i$ reaches a least fix-point. We define that fix-point as $\mathrm{NST}_{\mathcal{R}}$.

$$\mathrm{NST}_{\mathcal{R}} = \aleph_{\mathcal{R}}^k \ \textit{such that for all } m \geq k,\ \aleph_{\mathcal{R}}^m = \aleph_{\mathcal{R}}^k\ .$$

Example 36. *Consider the TRS $\mathcal{R}_{nest}$. The locations in $\mathrm{ENC}_{\mathcal{R}_{nest}}$ are underlined.*

$$\mathsf{st}(x) \to \mathsf{f}(\underline{\mathsf{a}(x)}) \quad \mathsf{f}(x) \to \mathsf{g}(\underline{\mathsf{a}(x)}) \quad \mathsf{g}(x) \to \mathsf{h}(\underline{\mathsf{a}(x)}) \quad \mathsf{h}(x) \to \mathsf{d}(\underline{x}, \underline{x}) \quad \mathsf{a}(x) \to \mathsf{0}$$

Here, the right-hand side of the previous rule is matched against the left-hand side of the next rule, resulting in more and more deeper nesting of redexes. However, it is not infinitely looping. We show now how the nesting depth of each of the underlined locations changes over each iteration.

- $\aleph^0_{\mathcal{R}_{nest}}$: $\mathsf{st}(x) \to \mathsf{f}(\underline{\overbrace{\mathsf{a}(\overbrace{x}^{0})}^{1}})$ $\mathsf{f}(x) \to \mathsf{g}(\underline{\overbrace{\mathsf{a}(\overbrace{x}^{0})}^{1}})$ $\mathsf{g}(x) \to \mathsf{h}(\underline{\overbrace{\mathsf{a}(\overbrace{x}^{0})}^{1}})$ $\mathsf{h}(x) \to \mathsf{d}(\overset{0}{\underline{x}}, \overset{0}{\underline{x}})$ $\mathsf{a}(x) \to \mathsf{0}$
- $\aleph^1_{\mathcal{R}_{nest}}$: $\mathsf{st}(x) \to \mathsf{f}(\underline{\overbrace{\mathsf{a}(\overbrace{x}^{0})}^{1}})$ $\mathsf{f}(x) \to \mathsf{g}(\underline{\overbrace{\mathsf{a}(\overbrace{x}^{1})}^{2}})$ $\mathsf{g}(x) \to \mathsf{h}(\underline{\overbrace{\mathsf{a}(\overbrace{x}^{1})}^{2}})$ $\mathsf{h}(x) \to \mathsf{d}(\overset{1}{\underline{x}}, \overset{1}{\underline{x}})$ $\mathsf{a}(x) \to \mathsf{0}$
- $\aleph^2_{\mathcal{R}_{nest}}$: $\mathsf{st}(x) \to \mathsf{f}(\underline{\overbrace{\mathsf{a}(\overbrace{x}^{0})}^{1}})$ $\mathsf{f}(x) \to \mathsf{g}(\underline{\overbrace{\mathsf{a}(\overbrace{x}^{1})}^{2}})$ $\mathsf{g}(x) \to \mathsf{h}(\underline{\overbrace{\mathsf{a}(\overbrace{x}^{2})}^{3}})$ $\mathsf{h}(x) \to \mathsf{d}(\overset{2}{\underline{x}}, \overset{2}{\underline{x}})$ $\mathsf{a}(x) \to \mathsf{0}$
- $\aleph^3_{\mathcal{R}_{nest}}$: $\mathsf{st}(x) \to \mathsf{f}(\underline{\overbrace{\mathsf{a}(\overbrace{x}^{0})}^{1}})$ $\mathsf{f}(x) \to \mathsf{g}(\underline{\overbrace{\mathsf{a}(\overbrace{x}^{1})}^{2}})$ $\mathsf{g}(x) \to \mathsf{h}(\underline{\overbrace{\mathsf{a}(\overbrace{x}^{2})}^{3}})$ $\mathsf{h}(x) \to \mathsf{d}(\overset{3}{\underline{x}}, \overset{3}{\underline{x}})$ $\mathsf{a}(x) \to \mathsf{0}$

At this point it holds $\aleph^3_{\mathcal{R}_{nest}} = \mathrm{NST}_{\mathcal{R}_{nest}}$.

Therefore, an expanded encoding $\widehat{\Phi}(\mathcal{R}_{nest})$ based on these results may return $\mathcal{R}'_{nest}$:

$$\mathsf{st}(x) \to \mathsf{f}(\mathsf{i}(l_{\mathsf{a}}(x))) \quad \mathsf{f}(x) \to \mathsf{g}(\mathsf{i}^2(l_{\mathsf{a}}(x))) \quad \mathsf{g}(x) \to \mathsf{h}(\mathsf{i}^3(l_{\mathsf{a}}(x))) \quad \mathsf{h}(x) \to \mathsf{d}(\mathsf{i}^3(x), \mathsf{i}^3(x)) \quad \mathsf{a}(x) \to \mathsf{0}$$

This gives us $\widehat{\Phi}(\mathcal{R}_{nest}) = \mathcal{R}'_{nest} \cup \{\mathtt{OMIT}, \mathtt{EXE}_{\mathsf{a}}, \mathtt{PROP}_{\mathsf{a}}, \mathtt{PROP}_{\mathsf{d}}\}$, where $\mathtt{PROP}_{\mathsf{a}}$ and $\mathtt{PROP}_{\mathsf{d}}$ are terminating.

Definition 37. *Let $\mathcal{R}$ be a TRS. We define* ***the expanded encoding*** *as a function $\widehat{\Phi}$:*

$$\widehat{\Phi}(\mathcal{R}) = \mathcal{R}' \cup \mathcal{S}, \text{ such that}$$

$$\begin{aligned}
\mathcal{R}' &= \{\widehat{\psi}(\alpha) \mid \alpha \in \mathcal{R}\},\\
\mathcal{S} &= \{\mathsf{i}(x) \xrightarrow{0} x\}\\
&\cup \{\, \mathsf{i}(l_f(\mathsf{x}_1, \ldots, \mathsf{x}_n)) \xrightarrow{0} \mathsf{i}(f(\mathsf{x}_1, \ldots, \mathsf{x}_n)) \quad \mid f \in \Sigma^n_d\}\\
&\cup \{\, \mathsf{i}(d\ (\mathsf{x}_1, \ldots, \mathsf{x}_n)) \xrightarrow{0} d(\mathsf{i}(\mathsf{x}_1), \ldots, \mathsf{i}(\mathsf{x}_n)) \quad \mid d \in \Sigma^n_c\}\\
&\cup \left\{ \mathsf{i}(l_f(\mathsf{x}_1, \ldots, \mathsf{x}_n)) \xrightarrow{0} \mathsf{i}(l_f(\mathsf{i}(\mathsf{x}_1), \ldots, \mathsf{i}(\mathsf{x}_n))) \;\middle|\; \begin{array}{l} f = \mathrm{root}(\mathcal{R}|_\lambda) \in \Sigma_d, \lambda \in \mathrm{ENC}_{\mathcal{R}},\\ \textit{for some } \mu \in \mathrm{INF}_{\mathcal{R}}, \mathrm{root}(\mathcal{R}|_\mu) = f \end{array} \right\}\\
&\cup \left\{ \mathsf{i}(l_f(\mathsf{x}_1, \ldots, \mathsf{x}_n)) \xrightarrow{0} l_f(\mathsf{i}(\mathsf{x}_1), \ldots, \mathsf{i}(\mathsf{x}_n)) \;\middle|\; \begin{array}{l} f = \mathrm{root}(\mathcal{R}|_\lambda) \in \Sigma_d, \lambda \in \mathrm{ENC}_{\mathcal{R}},\\ \textit{for all } \mu \in \mathrm{INF}_{\mathcal{R}}, \mathrm{root}(\mathcal{R}|_\mu) \neq f \end{array} \right\},
\end{aligned}$$

$\widehat{\psi}$ encodes the rules in $\mathcal{R}$ and uses $\widehat{\varphi}$ to encode the individual sides of each rule. We define the $\widehat{\psi}$-function as follows:

$$\widehat{\psi}(\alpha) = \widehat{\varphi}(\mathcal{R}, (\alpha, \mathtt{L}, \varepsilon)) \to \widehat{\varphi}(\mathcal{R}, (\alpha, \mathtt{R}, \varepsilon)), \;\; \alpha \in \mathcal{R}$$

For some location $\lambda = (\alpha, \mathtt{X}, \pi)$, $\mathtt{X} \in \{\mathtt{L}, \mathtt{R}\}$, $(\lambda, m) \in \mathrm{NST}_{\mathcal{R}}$ and $\mathcal{R}|_\lambda = f$, let $\lambda^{(k)} = (\alpha, \mathtt{X}, \pi.k)$, $k \in \mathbb{N}$. If $f \notin \mathcal{V}$, let $f \in \Sigma^n$.

1. $$\widehat{\varphi}(\mathcal{R}, \lambda) = \begin{cases} f\,(\;\widehat{\varphi}(\mathcal{R}, \lambda^{(1)}), \ldots, \;\widehat{\varphi}(\mathcal{R}, \lambda^{(n)})) & \textit{if} \quad f \in \Sigma_c \textit{ or } \lambda \notin \mathrm{ENC}_{\mathcal{R}}\\ l_f\,(\;\widehat{\varphi}(\mathcal{R}, \lambda^{(1)}), \ldots, \;\widehat{\varphi}(\mathcal{R}, \lambda^{(n)})) & \textit{else if } \mathtt{X} = \mathtt{L}\\ \mathsf{i}\;\;(l_f(\widehat{\varphi}_{\mathcal{N}}(\mathcal{R}, \lambda^{(1)}), \ldots, \widehat{\varphi}_{\mathcal{N}}(\mathcal{R}, \lambda^{(n)}))) & \textit{else if } \lambda \in \mathit{INF}_{\mathcal{R}}\\ \mathsf{i}^m(l_f(\widehat{\varphi}_{\mathcal{N}}(\mathcal{R}, \lambda^{(1)}), \ldots, \widehat{\varphi}_{\mathcal{N}}(\mathcal{R}, \lambda^{(n)}))) & \textit{else if } f \notin \mathcal{V}\\ \mathsf{i}^m(f) & \textit{else} \end{cases}$$

2. $$\widehat{\varphi}_{\mathcal{N}}(\mathcal{R},\lambda) = \begin{cases} f(\widehat{\varphi}(\mathcal{R},\lambda^{(1)}),\ldots,\widehat{\varphi}(\mathcal{R},\lambda^{(n)})) & \text{if} \quad f \in \Sigma_c \\ l_f(\widehat{\varphi}(\mathcal{R},\lambda^{(1)}),\ldots,\widehat{\varphi}(\mathcal{R},\lambda^{(n)})) & \text{else if } f \in \Sigma_d \\ f & \text{else} \end{cases}$$

Theorem 38. *Let $\mathcal{R}$ be a TRS and $\Phi(\mathcal{R})$ its encoding. Then* $\mathrm{rc}_{\mathcal{R}} = \mathrm{irc}'_{\widehat{\Phi}(\mathcal{R})}$.

Unfortunately, there was not enough time to prove Thm. 38. However, it was also manually tested, which showed that it can derive the same results as the regular encoding without the addition of non-terminating propagation rules.

5. Conclusion

In conclusion, this thesis has presented a novel technique for the automatic complexity analysis of TRSs. This new technique consists of encoding the TRS, so as to allow for other more powerful tools to analyze it. We have shown that this encoding fulfills the necessary theoretical conditions to be useful in practice. No tests were conducted via an established tool for automatic complexity analysis, because of the inability to limit the starting terms to the ones over the signature of the original TRS. Manual testing with the encoding showed the potential to improve the analysis on upper bounds.

An unproven refinement of the encoding has also been presented, which aims to improve the automatic analysis by limiting the addition of rules, which can lead to infinite rewrite sequences.

An implementation separate from any existing tools was developed, which given some TRS as input, returns the encoded version of it. Both, the standard and expanded encoding are implemented and readily available.

The concept of optional innermost rewriting and its relation to full rewriting (Lemma 24) was introduced, but not proven. Further, the future proof of the expanded encoding can also make use of Alg. 27. In terms of theoretical future work, there is also potential for the improvement of the over-approximation, namely in discovering non-ndg locations and locations with infinite nesting depth.

When it comes to practical work, implementing the technique as a processor in AProVE and extending it to respect the given start term restriction, are still open issues.

Bibliography

[1] G. Moser, "Proof theory at work: Complexity analysis of term rewrite systems," 2009.

[2] M. Avanzini, G. Moser, and M. Schaper, "TcT: Tyrolean Complexity Tool," in *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, vol. 9636. Springer Verlag Heidelberg, 2016, pp. 407–423.

[3] F. Frohn, J. Giesl, J. Hensel, C. Aschermann, and T. Ströder, "Lower bounds for runtime complexity of term rewriting," *Journal of Automated Reasoning*, vol. 59, pp. 1–43, 06 2017.

[4] F. Frohn and J. Giesl, "Analyzing runtime complexity via innermost runtime complexity," in *LPAR-21. 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning*. EasyChair, 2017.

[5] Termination competition. [Online]. Available: https://termination-portal.org/wiki

[6] F. Baader and T. Nipkow, *Term Rewriting and All That*. Cambridge University Press, 1998. [Online]. Available: https://www.cambridge.org/core/books/term-rewriting-and-all-that/71768055278D0DEF4FFC74722DE0D707

[7] J. Pol, van de and H. Zantema, "Generalized innermost rewriting," in *Rewriting Techniques and Applications (Proceedings 16th International Conference, RTA 2005, Nara, Japan, April 19-21, 2005)*, ser. Lecture Notes in Computer Science, J. Giesl, Ed. Germany: Springer, 2005, pp. 2–16.

[8] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke, "Mechanizing and improving dependency pairs," *J. Autom. Reasoning*, vol. 37, pp. 155–203, 10 2006.

[9] S. Lechner, "Data flow analysis for integer term rewrite systems," *RWTH*, 2022.