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

Coming soon

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

Supervisor
Stefan Dollase, M.Sc.

Statutory Declaration in Lieu of an Oath

Abstract

Shortly state the significance of the topic. Why is it relevant to analyze the full runtime complexity of term rewriting?

The automatic tools used to analyze full runtime complexity (rc) of term rewrite systems (TRSs) are still lacking as is shown in the results of the Termination Competition. Meanwhile upper bounds for innermost runtime complexity (irc) are much easily derived. This creates an opportunity to use techniques for irc to infer upper bound on rc. An encoding of the TRS, whose irc coincides with the rc of the original TRS, fills exactly this gap. The technique is implemented in the tool AProVE and tested sufficiently to show it improves the overall automatic complexity analysis.

This statement is a bit to strong and unprecise. Rather state that analyzing irc is stronger than analyzing rc.

Mention that the encoded system is relative.
The implementation is standalone, aka, not part of AProVE.

The technique was formalized, partly proven, and implemented.

The implementation was manually tested with some examples, to show that it finds a better runtime complexity.

Avoid introducing abbreviations in the abstract.

Contents

1. Introduction

Term rewrite systems(TRSs) offer a theoretical foundation to which computer programs can be abstracted to and analyzed. One of the topics of research concerning TRSs has to deal with the worst-case lengths of rewrite sequences, more commonly knows as complexity. In its analysis a distinction is made based on the starting term of the rewrite sequence: *derivational complexity*, considers any term, and *runtime complexity*, 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 rc, 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.

known

introduce it

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 on 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 discover by at least one of the participants in only 345 cases (36%).

The technique in [4] ~~over-approximates~~ 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, which this thesis aims to fill. This is done by extending the technique from [4] by encoding non-ndg TRSs in a way that $\mathrm{rc}_{\mathcal{R}} = \mathrm{irc}_{\mathcal{R}'}$, where $\mathcal{R}'$ is the encoding of $\mathcal{R}$. The two main problems in defining the encoding are calculating the set of non-ndg positions in a TRS and the 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.

identifies

The structure of the thesis is as follows. Chapter 2 introduces some of the preliminary knowledge regarding TRSs. Chapter 3 deals with the problems mentioned above and lays the foundations on which the encoding is defined. In Chapter 4 the encoding is presented accompanied with a proof of its usefulness. Chapter 5 serves as a summary and a look at future work.

shortly repeat it

Clearly state the remaining problem:
Find an upper bound for the irc a relative TRS,
when considering only some basic terms as start terms
(i.e., start terms can only contain some constructors).

2. Preliminaries

In this chapter is introduced 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 is. 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 in the focus of this thesis. ~~Combining rules together creates a term rewrite system.~~ A term rewrite system is a set of rules.

Example 1 (Addition).

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

Example 1. shows the simple TRS $\mathcal{R}_1$ for the calculation of addition on natural numbers. 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. To enhance readability, it is denoted $\mathsf{f}^\mathsf{n}(\mathsf{x})$, when some function f is applied n-many times to an input x.

The base case would be represented in α_1, where $x + 0 = x$. In α_2 the TRS 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:

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

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 inductively defined as: 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 $\mathsf{add}(\mathsf{x}, \mathsf{y})$ or more complex ones like $\mathsf{add}(\mathsf{add}(\mathsf{x}, \mathsf{s}(\mathsf{s}(\mathsf{0}))), \mathsf{s}(\mathsf{0}))$.

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 $pos(s)$ and inductively defined as follows:

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

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

   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 $root(s)$. 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 $|s| := |pos(s)|$.

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

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

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

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 xs. It can be written as

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

The term resulting from $\sigma(s)$ is called an **instance** of s.

This does not define how t\sigma is evaluated. If t=f(x) and \sigma(x)=a, then t\sigma=f(a).
This seems to be missing, since the remaining preliminaries are rather detailed.

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(lhs) and r as the right-hand side(rhs). A **term rewrite system** is a finite set of such rewrite rules combined with a signature Σ.

1. Given the signature Σ of a TRS $\mathcal{R}$ we have
   - the **set of defined symbols** $\Sigma_d := \{root(\ell) \mid \ell \to r \in \mathcal{R}\}$
   - 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** (**red**ucible **ex**pression) 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.
   - A proper subterm t_i of a term t is **nested**, if the t is a redex and $root(t_i) \in \Sigma_d$

   This is non-standard, so it should not be part of the preliminaries.
5. Rules are also associated with a **cost** and a **condition**. For the purposes of this thesis all rules have no conditions and a cost of either 0 ,denoted $\ell \xrightarrow{0} r$, or 1, denoted $\ell \to r$.

   Rather don't talk about conditions at all.

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

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 $s \to_{\mathcal{R}}^m t$.
3. Only rules with cost 1 contribute to the length of a rewrite sequence, i.e. for $s \to_{\mathcal{R}}^m t$ 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 $s \xrightarrow{i}_\pi \cancel{r}$ t

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

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

Definition 8 (Derivation height [4]).

The **derivation height** $dh : \mathcal{T} \times 2^{\mathcal{T}\times\mathcal{T}} \to \mathbb{N} \cup \{\omega\}$ 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.

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

In the case of an infinitely long rewrite sequence starting with term s, it is denoted as $dh(s, \to) = \omega$.

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 $\mathrm{rc}_{\mathcal{R}} : \mathbb{N} \to \mathbb{N} \cup \{\omega\}$ and $\mathrm{irc}_{\mathcal{R}} : \mathbb{N} \to \mathbb{N} \cup \{\omega\}$, defined as

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

Since every $\xrightarrow{i}$-sequence can be viewed as a $\to$-sequence, it is clear that $\mathrm{irc}_{\mathcal{R}}(n) \leq \mathrm{rc}_{\mathcal{R}}(n)$. Therefore, an upper bound for $\mathrm{rc}_{\mathcal{R}}$ infers an upper bound for $\mathrm{irc}_{\mathcal{R}}$ and a lower bound for $\mathrm{irc}_{\mathcal{R}}$ infers a lower bound for $\mathrm{rc}_{\mathcal{R}}$.

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 0, terminating after one more rewrite. $\mathcal{R}_1$ also belongs to a class of system, for which $\mathrm{rc}_{\mathcal{R}} = \mathrm{irc}_{\mathcal{R}}$. This class was the focus of [4] and we aim to go beyond it in this thesis.

Definition 10 (Non-dup-generalized (ndg) rewrite step [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 $\tau \in pos(\ell)\backslash\{\varepsilon\}$ with $root(\ell|_\tau) \in \Sigma_d$, 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 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 duplication, especially ~~the one of defined symbols~~ the duplication of redexes.. It is obvious that the duplication of ~~function calls~~ redexes affects the complexity and is by definition unreachable via an innermost evaluation strategy.

The second condition regards nested ~~defined symbols in the input of a rewrite.~~ redexes whose root symbol is matched. This means a function call is ignored, which once again can affect the complexity in unexpected ways.

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

Example 11.

Here we have the simple non-ndg TRS $\mathcal{R}_2$:

$$\begin{array}{rcl} \mathsf{f(x)} & \to & \mathsf{dbl(g(x))} \\ \mathsf{dbl(x)} & \to & \mathsf{d(x,x)} \\ \mathsf{g(s(x))} & \to & \mathsf{g(x)} \end{array}$$

Rule α_2 is duplicating and while that alone is no reason to call this TRS non-ndg, analyzing some sequences it produces proves exactly that. The longest sequence using the basic term $\mathsf{f}(\mathsf{s}(\mathsf{0}))$ is

$$\mathsf{f}(\mathsf{s}(\mathsf{0})) \to_{\mathcal{R}_2} \mathsf{dbl}(\mathsf{g}(\mathsf{s}(\mathsf{0})))\to_{\mathcal{R}_2} \mathsf{d}(\mathsf{g}(\mathsf{s}(\mathsf{0})), \mathsf{g}(\mathsf{s}(\mathsf{0}))) \to^2_{\mathcal{R}_2} \mathsf{d}(\mathsf{g}(\mathsf{0}), \mathsf{g}(\mathsf{0})),$$

which is just one step longer than the innermost rewrite sequence

Also show an example where we have an asymptotic effect.

$$\mathsf{f}(\mathsf{s}(\mathsf{0})) \to_{\mathcal{R}_2} \mathsf{dbl}(\mathsf{g}(\mathsf{s}(\mathsf{0}))) \to_{\mathcal{R}_2} \mathsf{dbl}(\mathsf{g}(\mathsf{0})) \to_{\mathcal{R}_2} \mathsf{d}(\mathsf{g}(\mathsf{0}), \mathsf{g}(\mathsf{0})).$$

Of course, as the size of the starting term grows the difference between both rewrite sequences increases, but it is important to note that both are linear in terms of asymptotic notation.

While a non-ndg TRS $\mathcal{R}$ fails to satisfy $\mathrm{rc}_{\mathcal{R}} = \mathrm{irc}_{\mathcal{R}}$, it is possible that a TRS $\mathcal{R}'$ exists such that $\mathrm{rc}_{\mathcal{R}} = \mathrm{irc}_{\mathcal{R}'}$. Ideally it would also hold that every full rewrite sequence in $\mathcal{R}$ has an equally long innermost rewrite sequence in $\mathcal{R}'$ and vice versa. The approach taken in this thesis is to transform the original TRS $\mathcal{R}$ by changing some rules and adding new ones, turning it into an encoded version of itself.

This irc is not allowed to have our special constructors in the basic start term.
Explain and define this at some point.
Also show an example why this is needed.

I like that you use colors to make it easier for the reader to follow the example.
However remember, that when printing the thesis, it is usually printed in black and white, because it is a lot cheaper.
Also, some people cannot see a difference between certain colors, for example (green-red color blindness).
Thus, it is always a good idea to additionally have an alternative way of highlighting.
For example a red arrow in a diagram could be dashed while a green arrow could be dotted.
Also, a red text could be singly underline while a green text could be doubly underlined.

I feel like one layer of explanation is missing. In the introduction, you say that the details will be explained later.
Later there are indeed the details, but at no point you explain the general idea
(i.e., replace matched defined symbols by constructors, so the matching step becomes innermost
and serialize/deserialize redexes that are later duplicated using constructors,
so it is possible to defer the execution of the redex after its duplication).

3. Foundations of the 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 1.) and the limitation of non-terminating rules (Section 2.). To this effort the goal of the encoding is to ensure each innermost rewrite sequence in the encoded TRS $\mathcal{R}'$ has an equally long full rewrite sequence in the TRS $\mathcal{R}$ and vice versa, which then serves as proof of $\mathrm{rc}_{\mathcal{R}} = \mathrm{irc}_{\mathcal{R}'}$. 3.1 3.2 ...

As we introduce more and more of the encoding we can observe via an example, how it changes and further show why certain positions in a TRS are ignored or are included.

Introduce a leading example very early in the thesis. Then go back to it every now and then to illustrate what is happening.
To understand the idea of the encoding very early, it might be useful to show an encoded version of the leading example very early.
Right now I feel like the reader has to read a large portion of the thesis to even get an idea about what is going on, but this should be very easy for the reader.

Example 12.

Consider the simple TRS $\mathcal{Q}$ with variable x Add a source, if you did not find the example by yourself.

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

We can begin by encoding the positions in rules, which directly break ndg-criteria. In our case, they are all located in rule α_2. A variable is duplicated and a defined symbol can be matched ~~to~~ on the lhs. We colored them in red for better illustration. Encoding in any manner these positions in isolation is quite meaningless. On one hand, we cannot be certain without further analysis if the duplication is applied on reducible expressions or not, so as to affect ~~time~~ runtime complexity. And replacing the defined symbol a on the lhs with some new constructor would make certain rewrite sequence unreachable. In order to preserve them, we must observe which other positions flow into the already marked as non-ndg positions and mark them as well. In the end, all marked positions will be encoded.

3.1 Non-ndg locations

Before we establish how we mark 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 position in terms.

Definition 13 (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

$$\mathcal{L}_{\mathcal{R}} := \{(\alpha, \underset{\mathtt{L}}{\cancel{\mathtt{X}}}, \pi) \mid \alpha = \ell \to r \in \mathcal{R},\ \cancel{\mathtt{X} \in \{\mathtt{L}, \mathtt{R}\},}\ \pi \in \mathrm{pos}(\ell)\ \cancel{\text{if } \mathtt{X} = \mathtt{L}, \text{ else } \pi \in \mathrm{pos}(r)}\}$$

\cup { ... same for rhs ... }

- The set of all **sub-locations** of a location λ is defined as
$$\mathrm{loc}_{\mathcal{R}}(\lambda) := \{(\alpha, \mathtt{X}, \pi) \mid \lambda = (\alpha, \mathtt{X}, \tau),\ \tau \leq \pi,\ (\alpha, \mathtt{X}, \pi) \in \mathcal{L}_{\mathcal{R}}\},$$
- The set of all **return locations of a symbol** f is defined as
$$\mathrm{return\_loc}_{\mathcal{R}}(f) := \{(\alpha, \mathtt{R}, \pi) \mid \alpha = \ell \to r \in \mathcal{R},\ \mathrm{root}(\ell) = f,\ \pi \in \mathrm{pos}(r)\}$$

Always motivate why a definition is useful. At this point it is unclear for return_loc(...).
When it is first used, one has to go back to the definition here.
This is more the style of writing a program than writing a paper.
In programming, all related function are defined in the same file.
In a paper, a function is introduced just before its first usage.

Below, return_loc is used with a location as its argument.

- We also define an iteration on the set of return locations. Let $f \in \Sigma$, then:
  - $\text{return\_loc}^1_{\mathcal{R}}(f) = \text{return\_loc}_{\mathcal{R}}(f)$
  - $\text{return\_loc}^n_{\mathcal{R}}(f) = \text{return\_loc}^{n-1}_{\mathcal{R}}(f) \cup \{\text{return\_loc}_{\mathcal{R}}(\text{root}(\mathcal{R}|_\mu)) \mid \mu \in \text{return\_loc}^{n-1}_{\mathcal{R}}(f)\}$
  - Let $\text{return\_loc}^*_{\mathcal{R}}(f)$ be the ~~fix-point~~ least fixed point of $\text{return\_loc}^n_{\mathcal{R}}(f)$.
- 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. However, this will lead to less precision in our algorithm.

In the general case, one always has to use an approximation.
The CAP function is usually used to increase the precision of the approximation, when compared to just using the defined symbol as the approximation.
... So the framing that the CAP function leads to less precision is a bit surprising.

Definition 14 (CAP$_{\mathcal{R}}$ [8])**.**

For a given TRS $\mathcal{R}$ and some term q we define $\text{CAP}_{\mathcal{R}} : \mathcal{T} \to \mathcal{T}$ as follows:

CAP(c(f,f)) = c(CAP(f),CAP(f)) = c(x,x)
If c is a constructor and f is defined.

This wording does not work well with the recursive definition.

What happens if q contains a variable multiple times? Does each occurrence have to be replaced by a fresh variables?

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

Both CAP(f) might be replaced by the same variable, because the variable only needs to be unique in CAP(f).

Rather say that x is a fresh variable, which means that it does not occur anywhere else.

In other words, in the resulting term $\text{CAP}_{\mathcal{R}}(q)$ 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.

use \cref

Consider the TRS $\mathcal{Q}$ from example 4. and the term $q = \mathsf{g}(\mathsf{a}, \mathsf{a})$, which corresponds to the rhs of rule α_1. As we will discover later, the locations of both a-symbols flow into the non-ndg locations in rule α_2. We discover this flow by matching $\text{CAP}_{\mathcal{R}}(q) = g(x, y)$ to the lhs of α_2. While it would have matched regardless of the application of the $\text{CAP}_{\mathcal{R}}$-function, there are cases where this is not quite as obvious.

An example would be nice.

As we continue to the algorithm for marking non-ndg locations it~~s~~ 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 first of three sets used to define the encoding is one that contains all locations marked as non-ndg.

At this point, the reader does not know about the three sets, so this sentence is rather confusing.

Definition 15.

Let $\mathcal{R}$ be a TRS.

1. The **set of base non-ndg locations** $\mathcal{X}'_{\mathcal{R}}$ is the smallest set such that:
   - $\{(\alpha, \mathtt{L}, \tau) \mid \alpha = \ell \to r \in \mathcal{R}, \quad \tau \in pos(\ell) \backslash \{\varepsilon\}, \quad root(\ell|_\tau) \in \Sigma_d\} \subseteq \mathcal{X}'_{\mathcal{R}}$ (S1)
   - $\{(\alpha, \mathtt{R}, \tau) \mid \alpha = \ell \to r \in \mathcal{R}, \tau, \pi \in pos(r), \tau \neq \pi, r|_\tau = r|_\pi \in \mathcal{V}\} \subseteq \mathcal{X}'_{\mathcal{R}}$ (S2)

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

2. The **set of over-approximated non-ndg locations** $\mathcal{X}_{\mathcal{R}}$ is the smallest set such that:
   - $\mathcal{X}'_{\mathcal{R}} \subseteq \mathcal{X}_{\mathcal{R}}$
   - $\left\{ (\alpha, \mathtt{L}, \tau) \;\middle|\; \begin{array}{l} \alpha = \ell \to r \in \mathcal{R}, \tau \in pos(\ell), \pi \in pos(r), \\ \ell|_\tau = r|_\pi \in \mathcal{V} \end{array} \right\} \subseteq \mathcal{X}_{\mathcal{R}}, \text{ if } (\alpha, \mathtt{R}, \pi) \in \mathcal{X}_{\mathcal{R}}$ (S3)
   - $\left\{ (\alpha, \mathtt{R}, \tau) \;\middle|\; \begin{array}{l} \alpha = \ell \to r \in \mathcal{R}, \tau \in pos(r), \tau = \pi.\upsilon, \upsilon \neq \varepsilon \\ \beta = s \to t \in \mathcal{R}, \omega \in pos(s), \upsilon \nparallel \omega, \\ \exists \text{ MGU for } \text{CAP}_{\mathcal{R}}(r|_\pi) \text{ and } s \end{array} \right\} \subseteq \mathcal{X}_{\mathcal{R}}, \text{ if } (\beta, \mathtt{L}, \omega) \in \mathcal{X}_{\mathcal{R}}$ (S4)
   - $\left\{ (\alpha, \mathtt{R}, \tau) \;\middle|\; \begin{array}{l} \alpha = \ell \to r \in \mathcal{R}, \tau \in pos(r), \\ \beta = s \to t \in \mathcal{R}, \pi \in pos(t), \cancel{root(t|_\pi) \in \Sigma_d}, \\ \exists \text{ MGU for } \text{CAP}_{\mathcal{R}}(t|_\pi) \text{ and } \ell \end{array} \right\} \subseteq \mathcal{X}_{\mathcal{R}}, \text{ if } (\beta, \mathtt{R}, \pi) \in \mathcal{X}_{\mathcal{R}}$ (S5)

This is already implied, because root(\ell) is defined and CAP(t|_\pi) is unifiable with \ell.

Explain for S3: Do all occurrences of the variable on the rhs have to be marked? Are all of the occurrences of the variable on the lhs marked?

$\mathcal{X}_\mathcal{R}$ 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 lhs of rules, which flow directly into marked variable locations on the rhs of the same rules. The second set $(S4)$ tracks the flow of locations from rhs of rules to lhs of rules. And finally $(S5)$ contains all over-approximated reachable return locations of marked defined symbols at rhs of rules. We are not using the already present definition of return locations, since we can exclude some of these based on whether the marked location can actually be matched and are not taken at face value, purely on if the root symbols match.

Avoid colloquial language.

Avoid the abbreviations lhs and rhs. They are rather colloquial.

We can now go into further detail why each of these locations are marked as non-ndg. Since our encoding will be analyzed via its irc, it cannot have nested defined symbols on the lhs of any rule. Such defined symbols 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 backwards 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 lhs of the rules in $(S2)$ to mark their origin. Afterwards, $(S4)$ checks if any subterm at a rhs of any rule can be matched to the marked locations in $(S1-\cancel{2})$. S2

This is also possible in innermost rewriting, e.g., start -> f(g(a)) g(b) -> stop f(g(x)) -> c(x) yields the innermost sequence start -> f(g(a)) -> c(a). We only need to prevent a redex to occur below the redex position. To forbid all defined symbols below the root in the lhs is sufficient, but not necessary.

S5

Since $(S4-\cancel{5})$ are more complex compared to the rest, we will go into further detail for each step. In $(S4)$ the position τ is split in 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 rhs 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 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 with 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 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 rhs of β_1 does not flow into position 1 of the lhs of β_2. Using the CAP$_\mathcal{R}$ function on $\mathsf{g}(\mathsf{h})$ returns $\mathsf{g}(x)$, which can be matched.

Lastly, we require that $\upsilon \nparallel \omega$. This part ensures that the newly included location in α actually flows into a non-ndg location. In example 12. $\lambda_1 := (\alpha_1, \mathtt{R}, 1)$ flows only into $\lambda_2 := (\alpha_3, \mathtt{L}, 1)$ and we would want to include λ_1, only if λ_2 is non-ndg.

In the given example one could also use = instead of \nparallel. => Illustrate the edge cases?

\alpha_2 ?

In $(S5)$ are included the return locations of non-ndg marked defined symbols. The subterms on rhs of rules, which have a defined root symbol, may be redexes and thus rewritten. The newly evaluated term at the position of rewriting must inherit the non-ndg mark, 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 example 12, we can now see how the newly defined set helps in establishing the set of locations to be encoded. Here the base non-ndg locations $((S1-2))$ are colored in red, just as they were in the beginning of the chapter. Furthermore, the locations in $\mathcal{X}_\mathcal{Q}$ are underlined in green.

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

The variable at $(\alpha_2, \mathtt{L}, 1)$ is now marked, since its being duplicated when rewritten. Let us now take a closer look at the rhs of α_1. The a-symbols at positions 1 and 2 flow into non-ndg marked locations in α_2. In this case there is no difference if they flow into a variable or another defined symbol. The a-symbol at position 3 however, flows into the variable y, which gets deleted when rewritten. Therefore, it does not flow into a non-ndg location and is not marked.

Lastly, we also notice that the constructor b in rule α_3 has been marked. This location is considered a return position of a and by $(S5)$ it is marked. But because its root is a constructor and therefore not

It is a bit confusing to explain the encoding and the flow analysis at the same time.

a redex, we have no use to encode that location. For the same reason, we would have no use to encode 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 $\mathrm{rc}_{\mathcal{Q}}$, 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 rhs of rules, will then allow us to define all non-ndg marked locations, which can potentially receive a redex.

Definition 16.

For some given TRS $\mathcal{R}$ and a set of locations Δ let $\mathcal{Y}_{\mathcal{R}}^{\Delta}$ be the smallest set containing Δ and all locations, which locations from Δ might flow into:

- $\Delta \subseteq \mathcal{Y}_{\mathcal{R}}^{\Delta}$ $\quad (S6)$

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

Maybe have S8 before S7, to have the same order as for S3 and S4?

Replace \tau by \pi and vice versa? This makes it easier to see the difference to S3.

- $\left\{ (\alpha, \mathtt{R}, \tau) \;\middle|\; \begin{array}{l} \alpha = \ell \to r \in \mathcal{R}, \pi \in pos(\ell), \tau \in pos(r), \\ \ell|_{\pi} = r|_{\tau} \in \mathcal{V} \end{array} \right\} \subseteq \mathcal{Y}_{\mathcal{R}}^{\Delta}, \text{ if } (\alpha, \mathtt{L}, \pi) \in \mathcal{Y}_{\mathcal{R}}^{\Delta}$ $\quad (S8)$

Explain: Why is the equivalent of S5 missing?

The definitions of these subsets are similar to $\underline{(S3-4)}$. Changed is which location from a data flow is included.

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

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

where the locations in $\mathcal{Y}_{\mathcal{Q}}^{\Delta}$ are indicated with ˆ ('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 marked constructor in α_3, as was desired. Further, even though the a-symbol at $(\alpha_1, \mathtt{R}, 3)$ has a 'hat' it does not flow into a non-ndg location, therefore encoding it is not necessary.

One more location of interest in that intersection is the variable x at $(\alpha_2, \overset{\mathtt{L}}{\cancel{\mathtt{R}}}, 1)$. It does not make sense to encode variables on the lhs, so they should be excluded from the final set too.

Definition 17($\mathrm{ENC}_{\mathcal{R}}$)**.**

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

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

Going back to TRS $\mathcal{Q}$ and highlighting only the locations in $\mathrm{ENC}_{\mathcal{Q}}$, we get

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

The next section discusses the concept of the encoding and what it means for the marked locations to be encoded. A more precise definition of the encoding is then presented in the next chapter.

3.2 Concept of the encoding

Continuing with TRS $\mathcal{Q}$ and its over-approximated non-ndg location, we can now start looking into how to encode them. 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}.

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

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.

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

Replacing defined symbols with constructor creates a new problem. There are rewrite sequences which evaluate these exact term, that we have now practically removed. In our example 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.

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

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

Again, we run into a problem of matching rhs of α_1 to lhs 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 anyways. In situations like this we would like to remove the i-symbols from the redex, so the rewrite is innermost.

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

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

After all these changes, the duplication in α_2 has remain unaltered. We will show now why the marked x's should be encapsulated by i, similar to the rhs of α_1. Consider the following rewrite sequence using the ~~rule~~ rules $\alpha_1 - \alpha_5$ directly above.

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

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 marked variable positions. Therefore an i-symbol should be placed there when rewriting with α_2.

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

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

We get the final version of TRS $\mathcal{Q}$, which corresponds to the eventual encoding of $\mathcal{Q}$, except one insignificant change in α_4. There is however more to these rules than is observed here with this simple example. The next section discusses the topic of what rules the encoding should add and introduces a new problem, namely the inclusion of non-terminating rewrite rules.

3.3 Rules

The additional rules exclusively evaluate redexes with the newly introduced defined symbol i at their root and can be split into three categories.

- $\mathsf{i}(l_\mathsf{a}) \xrightarrow{0} \mathsf{i}(\mathsf{a})$ f and a should be the same.

  The *executing rules* restore the defined symbol, whose constructor version is at position 1 on the lhs. Shorthand notation for this rule is $\texttt{EXE}_f$, where f 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 OMIT.

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

  g and f should be the same.

  The *propagation rule* encapsulates with an i-symbol all subterms directly bellow the root of the term matched at position 1. The example shown above assumes was missing from the concept in the last section, because it was indeed not needed. We know this because there is no rewrite sequence that requires the propagation of any i-symbols. Shorthand notation for this rule is $\mathtt{PROP}_f$, where f is the symbol at which the propagation occurs.

  What does that mean?

  propagation?

  The omission rules have two variations:

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

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

  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 $\mathcal{R}'$ creates non-terminating rewrite sequences like the following:

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

  Rather say that it is currently unknown how to analyze the irc of a relative TRS with non-terminating relative rules.

  Such sequences could throttle the automatic complexity analysis and therefore it is best to avoid adding non-terminating rules whenever possible. An analysis on the rules for which function symbols it would be viable to add the terminating version of the rule, is provided in this section.

For each of these shorthand notation, we use a superscript L or R to indicate lhs or rhs of the respective rule. For example $\mathtt{OMIT}^L = \mathsf{i}(x)$.

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.

The first and easiest condition to start with is checking if any nesting of encoded defined symbols occurs in any of the rules. If that is not the case, then any propagation of the i-symbol would encapsulate terms in normal form. Then all propagation rules would not be productive and therefore be omitted. This is however rarely the case, but is a step in the right direction.

What does it mean?

Example 18.

Consider the following TRS $\mathcal{Z}$ with variables x and y. The locations in $\mathrm{ENC}_{\mathcal{Z}}$ are highlighted.

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

We now encounter something that was missing in the previous example - locations of nested defined symbols marked as non-ndg. One intuitive approach to encode these locations would be like so:

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

Call the TRS above $\mathcal{Z}'$ and consider the rewrite sequence

$$\mathsf{h} \quad \to_{\mathcal{Z}'} \quad \mathsf{g}(\mathsf{i}(l_{\mathsf{a}}(\mathsf{i}(l_{\mathsf{b}}))), \mathsf{i}(l_{\mathsf{c}}(\mathsf{i}(l_{\mathsf{b}})))) \quad \xrightarrow{0}{}^{*}_{\mathtt{OMIT}} \quad \mathsf{g}(l_{\mathsf{a}}(l_{\mathsf{b}}), l_{\mathsf{c}}(l_{\mathsf{b}})) \quad \to_{\mathcal{Z}'} \quad \mathsf{f}(\cdots,\ \mathsf{i}(l_{\mathsf{a}}(l_{\mathsf{b}})),\ \mathsf{i}(l_{\mathsf{c}}(l_{\mathsf{b}})))$$

We can ignore the subterm at position 1 in the last term, since we are interested in the other two positions. In the original TRS $\mathcal{Z}$ the b terms nested at positions 2.1 and 3.1 of a similar rewrite sequence, can be reduced to $\mathsf{0}$. In $\mathcal{Z}'$ that would only be possible if the i-symbols could be propagated deeper, so as to restore the b-symbols. Therefore, the encoding has to add the rule $\mathtt{PROP}_{\mathsf{a}}$ and also does not need to add an i-symbol in front of each marked location with a defined symbol. It suffices to put only one in front of the outermost marked location and include the non-terminating propagation rules for the respective symbols. However, if want to add the terminating version of these rules instead, we need to analyze the TRS further.

The very nesting of defined symbols would not be problematic, if it is of constant size depth ~~i.e.~~ , i.e., there is a maximum for the number of defined symbols that can be nested at a given location in any rewrite sequence of a given TRS. In such cases the encoding can add multiple i-symbols at the location of these nestings instead of just one. This allows for the addition of terminating propagation rules, since each of the n-many i-symbols at the encoded location can be propagated to evaluate one of the n-many nested redexes.

Show an example where the nesting depth is not finite.

This is supposed to _over-approximate_ the nesting depth of terms in that location.

Definition 19.

For a given TRS $\mathcal{R}$ the **nesting size** of a location $\lambda \in \mathcal{L}_\mathcal{R}$ is defined as the result of the function $nest_\mathcal{R} : \mathcal{L}_\mathcal{R} \to \mathbb{N}$. Let $max\varnothing = 0$.

$$nest_\mathcal{R}(\lambda) = \begin{cases} max\{nest_\mathcal{R}(\mu) | \mu \in loc_\mathcal{R}(\lambda)\backslash\{\lambda\}\} & \text{if } root(\mathcal{R}|_\lambda) \notin \Sigma_d \\ 1 + max\{nest_\mathcal{R}(\mu) | \mu \in loc_\mathcal{R}(\lambda)\backslash\{\lambda\}\} & \text{else} \end{cases}$$

Going back to TRS $\mathcal{Z}$ and measuring the nesting size indicated here with the braces over the marked locations, we get the following result:

$$\begin{array}{lll} \alpha_1 : \mathsf{h} \to \mathsf{g}(\overbrace{\mathsf{a}(\mathsf{b})}^{2}, \overbrace{\mathsf{c}(\mathsf{b})}^{2}) & & \alpha_2 : \mathsf{g}(x, y) \to \mathsf{f}(\mathsf{g}(\overbrace{\mathsf{a}(x))}^{1}, \overbrace{\ y\ }^{0}), \overbrace{\ x\ }^{0}, \overbrace{\ y\ }^{0}) \\ \alpha_3 : \mathsf{a}(x) \to 0 & \alpha_4 : \mathsf{b} \to 0 & \alpha_5 : \mathsf{c}(x) \to 0 \end{array}$$

We can mend our encoding to adds as many i-symbols as the nesting size of the location. This will allow the propagation of i-symbols at the rhs of an encoded α_1, but would also not encode the three variables with nesting size 0 in α_2. Besides that, it is clear that a rewrite sequence would not be able to restore the defined symbols at these positions too. Therefore, tracking where each of these 'nests' flow should create a more accurate encoding.

Clarify, that def 19 was just a first draft and won't be used afterwards.

Definition 20 (NST$_\mathcal{R}$)**.**

A **nest** is a tuple, where the first element is a location λ in some TRS $\mathcal{R}$ and the second is its nesting size $nest_\mathcal{R}(\lambda)$.

1. For a TRS $\mathcal{R}$ let $\aleph_\mathcal{R}$ be the smallest set such that:
   - $\{(\lambda, n) \mid \lambda \in \mathcal{L}_\mathcal{R}, n = nest_\mathcal{R}(\lambda)\} \subseteq \aleph_\mathcal{R}$
   - $\{(\mu, n) \mid \mu \in \mathcal{Y}_\mathcal{R}^{\{\lambda\}}\} \qquad \subseteq \aleph_\mathcal{R}$, if $(\lambda, n) \in \aleph_\mathcal{R}$
2. We now define $\aleph'_\mathcal{R}$, which includes only one nest per location. We are also only interested in the one with the greatest nesting size.
   - ~~$\{(\lambda, n) \mid \forall(\lambda, m) \in \aleph_\mathcal{R}, n \geq m\}$~~ $=: \aleph'_\mathcal{R}$ { (\lambda, n) | n=max{m | (\lambda, m) \in N_R} }
3. Lastly we need to update the locations of their new nesting sizes according to the values in $\aleph'_\mathcal{R}$.
   - $$\text{nest}'_\mathcal{R}(\lambda) = \begin{cases} n & \text{if} & loc_\mathcal{R}(\lambda)\backslash\{\lambda\} = \varnothing \text{ and } (\lambda, n) \in \aleph'_\mathcal{R} \\ max\{\text{nest}'_\mathcal{R}(\mu) | \mu \in loc_\mathcal{R}(\lambda)\backslash\{\lambda\}\} & \text{else if} & root(\mathcal{R}|_\lambda) \notin \Sigma_d \\ 1 + max\{\text{nest}'_\mathcal{R}(\mu) | \mu \in loc_\mathcal{R}(\lambda)\backslash\{\lambda\}\} & \text{else} & \end{cases}$$
4. For a TRS $\mathcal{R}$ we define the set of actual nests in $\mathcal{R}$, affected by data flows, NST$_\mathcal{R}$ as

$$\text{NST}_\mathcal{R} := \{(\lambda, n) \mid \lambda \in \mathcal{L}_\mathcal{R}, n = nest'_\mathcal{R}(\lambda)\}$$

Let us now apply this to TRS $\mathcal{Z}$. We can disregard the results of the previous calculations. So the results from nest$'_\mathcal{R}$ are written in their place now in red.

If I see it correctly, then this approach only considers the nesting depth created by the current and the calling rule, but not any previous calling rules.

1: start(x) -> f(a(x))
2: f(x) -> g(a(x))
3: g(x) -> h(a(x))
4: a(x) -> stop

Now in location (3,R,1) the nesting depth should be 3.

The solution is probably to merge def 20 into a single large fixed point iteration.

Note that this does not terminate for infinite nestings. So you might want to run the INF analysis first and only apply the NST analysis on the remaining locations. Alternatively, you could merge the INF and NST analyses.

$$\alpha_1 : \mathsf{h} \to \mathsf{g}(\overbrace{\mathsf{a}(\mathsf{b})}^{2}, \overbrace{\mathsf{c}(\mathsf{b})}^{2}) \qquad \alpha_2 : \mathsf{g}(x, y) \to \mathsf{f}(\mathsf{g}(\overbrace{\mathsf{a}(x))}^{3}, \overbrace{\; y \;}^{2}), \overbrace{\; x \;}^{2}, \overbrace{\; y \;}^{2})$$

$$\alpha_3 : \mathsf{a}(x) \to 0 \qquad \alpha_4 : \mathsf{b} \to 0 \qquad \alpha_5 : \mathsf{c}(x) \to 0$$

Avoid the term relative, since it has a clear meaning in rewriting.

This solution works only in the cases where the nesting does not grow, i.e. the nests from α_1 stay constant relative to rewriting. However, the nest in α_2 with size 3 is a great example of such growing nests. Its location - $(\alpha_2, \mathtt{R}, 1.1)$ flows into the variable in the same rule - $(\alpha_2, \mathtt{L}, 1)$, which then flows into a location bellow the nest - $(\alpha_2, \mathtt{R}, 1.1.1)$. Regardless of the initial input of variable x this creates a positive feedback loop and with each rewrite step using α_2 the nest grows in size. Consider the rewrite sequence:

$$\mathsf{g}(0, 0) \quad \to^{n}_{\mathcal{Z}} \quad \underbrace{f(f(\cdots f}_{n-times}(\mathsf{g}(\mathsf{a}^{n}(0), 0),\ \mathsf{a}^{n-1}(0),\ 0), \cdots))$$

The wording is unclear.

There is no number of i-symbols the encoding can encapsulate the nest in $\mathcal{Z}$ so that it results in this, since it can always grow beyond it. The encoding has to be able to automatically find all such non-ndg marked locations and add the non-terminating propagation rule for their root symbol.

Definition 22 (INF$_{\mathcal{R}}$)**.**

For a TRS $\mathcal{R}$ let the **set of locations of repeatedly nesting defined symbols** INF$_{\mathcal{R}}$ be defined as the smallest set such that:

- $\{\lambda \mid \mu \neq \lambda,\ \mu \in loc_{\mathcal{R}}(\lambda),\ \mu \in \mathcal{Y}^{\{\lambda\}}_{\mathcal{R}}\} \subseteq \mathrm{INF}_{\mathcal{R}}$ $\quad (I1)$
- $\{\lambda \mid \mu \neq \lambda,\ \mu \in loc_{\mathcal{R}}(\lambda),\ \mathrm{root}(\mathcal{R}|_{\mu}) \in \Sigma_d\} \subseteq \mathrm{INF}_{\mathcal{R}},\ \text{if } \mu \in \mathrm{INF}_{\mathcal{R}}$ $\quad (I2)$
- $\{\lambda \mid \mu \in \mathrm{return\_loc}^*(\lambda),\ \mathrm{root}(\mathcal{R}|_{\lambda}) = \mathrm{root}(\mathcal{R}|_{\mu})\} \subseteq \mathrm{INF}_{\mathcal{R}}$ $\quad (I3)$
- $\{\lambda \mid \mu \in \mathrm{return\_loc}(\lambda)\} \subseteq \mathrm{INF}_{\mathcal{R}},\ \text{if } \mu \in \mathrm{INF}_{\mathcal{R}}$ $\quad (I4)$

Adding the equivalent of S5 to def 16 should make I3 and I4 unnecessary.

Above, return_loc is defined with a symbol as its argument.

In TRS $\mathcal{Z}$ from example 18. we can only observe a single location to be included in INF$_{\mathcal{Z}}$. It is included via set $(I1)$, which is also the one to make the most intuitive sense. As shown above, locations of defined symbols, which flow bellow themselves can cause repeated nesting. But that is also true if any of the subterms of a location cause repeated nesting. This is investigated by $(I2)$. Further, if the redex at the nest is rewritten by a rule of the form $\alpha : \mathsf{a}(x) \to \mathsf{a}(\mathsf{a}(x))$ it is obvious to us that this can grow infinitely. In fact, the location of the a-symbol on the rhs will be included in INF via $(I1)$, but a different location, for which that is a return location, would not be included, if it is not for $(I4)$. Another over-approximation introduced here is the one in $(I3)$. Without further analysis, we assume that if the same defined symbol occurs at any of the return locations, then a repeated nesting occurs. This is most noticeable, when dealing with 0 arity defined symbols. A rule $\beta : \mathsf{h} \to \mathsf{a}(\mathsf{h})$, for some $\mathsf{a} \in \Sigma_d$, does nest repeatedly, but would not be discovered by all other defined subsets. Not only that, but $(I4)$ would be stuck in an infinite loop determining if the h-symbol on the rhs is in INF.

Which one?

The order of these subsets has no impact on the theoretical definition of INF, but in practice, the algorithm should run starting from $(I1)$ and ending in $(I4)$. If there are any locations with the same defined symbol among the return positions, then the algorithm terminates, otherwise it continues to step $(I4)$, which cannot loop infinitely now.

The encoding should also add propagation rules for constructor symbols. This was not cover in any of the examples, but is quite trivial, because if there happens to be a constructor symbol bellow a non-ndg marked location and a encoded defined symbol bellow the constructor, 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.

For a non-ndg TRS $\mathcal{R}$ the sets ENC$_{\mathcal{R}}$, NST$_{\mathcal{R}}$ and INF$_{\mathcal{R}}$ form the base on which the encoded version $\mathcal{R}'$ is created. Before we continue to next chapter let us revisit each of these sets.

Isn't this term defined by ENC_R? So this explanation is self-referential.
Rather say, that ENC_R contains all locations that might contain a defined symbol
and that also might be duplicated or matched later.

- $\mathrm{ENC}_{\mathcal{R}}$ contains all locations marked as non-ndg and which potentially have a redex flow into them.
- $\mathrm{NST}_{\mathcal{R}}$ ~~keeps track of~~ the nesting size of each location in $\mathcal{R}$. over-approximates.
- $\mathrm{INF}_{\mathcal{R}}$ contains all locations of defined symbols, which ~~are proven to~~ might flow into one of its sub-locations.

4. Encoding

In this chapter, 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 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.

It is also useful to get a more precise lower bound for the worst-case.

Definition 23.

Let $\mathbb{T}$ be the set of all TRSs. We define **the encoding** as a function $\Phi : \mathbb{T} \to \mathbb{T}$:

This is usually use for the set of terms.

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

$$\begin{aligned}
\mathcal{R}' &= \{\psi(\alpha) \mid \alpha \in \mathcal{R}\} \\
\mathcal{S} &= \{\mathsf{i}(x) \xrightarrow{0} x\} \\
&\cup \{\, \mathsf{i}(l_f(\mathsf{x_1}, \ldots, \mathsf{x_n})) \xrightarrow{0} \mathsf{i}(f(\mathsf{x_1}, \ldots, \mathsf{x_n})) \mid f \in \Sigma_d^n\} \\
&\cup \{\, \mathsf{i}(d\,(\mathsf{x_1}, \ldots, \mathsf{x_n})) \xrightarrow{0} d(\mathsf{i}(\mathsf{x_1}), \ldots, \mathsf{i}(\mathsf{x_n})) \mid d \in \Sigma_c^n\} \\
&\cup \left\{ \mathsf{i}(l_f(\mathsf{x_1}, \ldots, \mathsf{x_n})) \xrightarrow{0} \mathsf{i}(l_f(\mathsf{i}(\mathsf{x_1}), \ldots, \mathsf{i}(\mathsf{x_n}))) \,\middle|\, \begin{array}{l} f = root(\mathcal{R}|_\lambda) \in \Sigma_d, \lambda \in \mathrm{ENC}_\mathcal{R}, \\ \exists \mu \in \mathrm{INF}_\mathcal{R}, root(\mathcal{R}|_\mu) = f \end{array} \right\} \\
&\cup \left\{ \mathsf{i}(l_f(\mathsf{x_1}, \ldots, \mathsf{x_n})) \xrightarrow{0} l_f(\mathsf{i}(\mathsf{x_1}), \ldots, \mathsf{i}(\mathsf{x_n})) \,\middle|\, \begin{array}{l} f = root(\mathcal{R}|_\lambda) \in \Sigma_d, \lambda \in \mathrm{ENC}_\mathcal{R}, \\ \forall \mu \in \mathrm{INF}_\mathcal{R}, root(\mathcal{R}|_\mu) \neq f \end{array} \right\}
\end{aligned}$$

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

as

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

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

f = root(R|_\lambda)

1.

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

I think \phi should rather receive a location as its argument.

2.

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

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 24.

Consider the following TRS $\mathcal{P}$ and all the relevant information about its encoding. The locations in $\mathrm{ENC}_{\mathcal{P}}$ are highlighted, the nesting sizes of the relevant location taken from $\mathrm{NST}_{\mathcal{P}}$ is listed above them, and the only location in $\mathrm{INF}_{\mathcal{P}}$ is underlined in rule α_3.

$$\alpha_1 : \mathsf{h} \to \mathsf{a}(\overbrace{\mathsf{f}(\mathsf{c})}^{2}) \qquad\qquad \alpha_2 : \mathsf{a}(x) \to \mathsf{a}(\overbrace{\underline{\mathsf{a}(x)}}^{1})$$

$$\alpha_3 : \mathsf{f}(\mathsf{c}) \to \overbrace{\mathsf{f}(\mathsf{c})}^{2} \qquad \alpha_4 : \mathsf{a}(x) \to \mathsf{b}(\overbrace{\ x\ }^{2}, \overbrace{\ x\ }^{2}) \qquad \alpha_5 : \mathsf{c} \to 0$$

Calculating the result of $\Phi(\mathcal{P})$ goes through the following process. Terms with zero non-ndg marked locations are skipped, since the first condition of φ clearly ignores these and changes nothing in the process.

- $\psi(\alpha_1) = \mathsf{h} \to \varphi(\mathsf{a}(\mathsf{f}(\mathsf{c}))) \quad = \mathsf{h} \to \mathsf{a}(\varphi(\mathsf{f}(\mathsf{c}))) \quad = \mathsf{h} \to \mathsf{a}(\mathsf{i}^2(l_{\mathsf{f}}(\varphi_{\mathcal{N}}(\mathsf{c})))) \quad = \mathsf{h} \to \mathsf{a}(\mathsf{i}^2(l_{\mathsf{f}}(l_{\mathsf{c}}))$
- $\psi(\alpha_2) = \mathsf{a}(x) \to \varphi(\mathsf{a}(\mathsf{a}(x))) = \mathsf{a}(x) \to \mathsf{a}(\varphi(\mathsf{a}(x))) = \mathsf{a}(x) \to \mathsf{a}(\mathsf{i}(l_{\mathsf{a}}(\varphi_{\mathcal{N}}(x)))) = \mathsf{a}(x) \to \mathsf{a}(\mathsf{i}(l_{\mathsf{a}}(x)))$
- $\psi(\alpha_3) = \varphi(\mathsf{f}(\mathsf{c})) \to \varphi(\mathsf{f}(\mathsf{c})) \ = \mathsf{f}(\varphi(\mathsf{c})) \to \mathsf{i}^2(l_{\mathsf{f}}(\varphi_{\mathcal{N}}(\mathsf{c}))) = \mathsf{f}(l_{\mathsf{c}}) \to \mathsf{i}^2(l_{\mathsf{f}}(l_{\mathsf{c}}))$
- $\psi(\alpha_4) = \mathsf{a}(x) \to \varphi(\mathsf{b}(x,x)) \ = \mathsf{a}(x) \to \mathsf{b}(\varphi(x), \varphi(x)) \quad = \mathsf{a}(x) \to \mathsf{b}(\mathsf{i}^2(x), \mathsf{i}^2(x))$
- $\psi(\alpha_5) = \alpha_5$

Describing every step in detail is pointless, but in a few words we can summarize the process of encoding the rules of $\mathcal{P}$ as follows. The subterm on the lhs of α_2 contains a constant size nest that flows into the duplication of α_4 and thus is encapsulated by two i-symbols. A repeated nesting has been discovered in the next rule α_2. Therefore, only a single i-symbol is placed in front of it. In α_3 is our only case of nested defined symbol on a lhs in TRS $\mathcal{P}$. It is only replaced by a fresh constructor symbol and the entire rhs of α_3 is encoded, since these are return locations of f, marked non-ndg in α_1. The rhs of α_4 contains the duplicated variables, which actually have a nesting size of 2 due to the nest from α_1. Next come the additional rules.

- $\mathcal{S} = \{\mathsf{i}(x) \to x\} \cup \cdots$

Example not finished :((

In order to prove $\mathrm{rc}_{\mathcal{R}} = \mathrm{irc}_{\Phi(\mathcal{R})}$ we need to define a new type of rewrite relation. The reason is that we discovered a major problem, when first attempting to prove the $\subseteq$ direction of the equality. 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. One idea was to extend Lemma 8. from [7] to relative TRSs. Briefly, Lemma 8. states that for each ndg rewrite sequence of length n starting with t, there is also an innermost rewrite sequence of length n starting with t. Assuming we could prove this statement, 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 counterexample was discovered using a non-terminating rewrite sequence. Excluding them, 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, but make it so that when constructing the rewrite sequence in the encoded TRS, we would not need to change the order of rule application. The new rewrite relation is based on a tagging, which works similarly to the calculating of non-ndg locations, but cuts the over-approximation.

Definition 24(Optional innermost) **.**

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

$$\tau \gneqq \pi \text{ such that } t_i|_{\tau} \notin \mathrm{NF} \ \Rightarrow \tau \in \mathit{tag}_{\nabla}(t_i) \ .$$

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

Definition 25.(tag_∇)

1. Given a *terminating* rewrite sequence $\nabla = t_0 \rightarrow_{\alpha_1,\pi_1} t_1 \rightarrow \cdots \rightarrow_{\alpha_n,\pi_n} t_n$ in a TRS $\mathcal{R}$ we define $tag_\nabla(t_i) \subseteq pos(t_i)$ as the smallest such that:
   - $\{\tau \mid \tau = \pi_i.\upsilon, \quad (\alpha_i, \mathtt{R}, \omega) \in \mathcal{X}'_{\mathcal{R}}, \omega \leq \upsilon\} \;\subseteq tag_\nabla(t_i)$
   - $\{\tau \mid \tau = \pi_{i+1}.\upsilon, (\alpha_i, \mathtt{L}, \omega) \in \mathcal{X}'_{\mathcal{R}}, \omega \leq \upsilon\} \;\subseteq tag_\nabla(t_i)$
   - $\left\{ \tau \;\middle|\; \begin{array}{l} \alpha_{i+1} = \ell_{i+1} \rightarrow r_{i+1}, \\ \omega_L \in pos(\ell_{i+1}), \quad \omega_R \in pos(r_{i+1}), \\ x = \ell_{i+1}|_{\omega_L} = r_{i+1}|_{\omega_R} \in \mathcal{V}, \\ \pi_{i+1}.\omega_L.\tau_x = \tau, \quad \pi_{i+1}.\omega_R.\tau_x = \gamma \end{array} \right\} \subseteq tag_\nabla(t_i)$, if $\gamma \in tag_\nabla(t_{i+1})$
2. Given a *non-terminating* rewrite sequence$\nabla = t_0 \rightarrow_{\alpha_1,\pi_1} t_1 \rightarrow \cdots$, let ∇^n be the first n steps of ∇. Then there exists $m \in \mathbb{N}$ such that:

$$tag_\nabla(t_i) = \bigcup_{k=0}^{m} tag_{\nabla^k}(t_i).$$

As mentioned above, there are cases where the rule application order in a full rewrite sequence in $\mathcal{R}$ cannot be maintained when constructing an innermost rewrite sequence in $\Phi(\mathcal{R})$. However, this is not true for oi rewrite sequences in $\mathcal{R}$. Therefore, the first step in proving $\mathrm{rc}_\mathcal{R} \subseteq \mathrm{irc}_{\Phi(\mathcal{R})}$ is to show that full rewrite sequences of length n starting with term t imply the existence of oi rewrite sequence of the same length and starting term.

Lemma 26.

For some TRS $\mathcal{R}$, if $t \rightarrow^n_\mathcal{R} v$ then $t \xrightarrow{oi}{}^{n}_{\mathcal{R}} u$ for some term u.

Now we can take the oi rewrite sequences and use them as input for the construction of the innermost rewrite sequence in the respective encoded TRS. An algorithm for this task is presented later. Before we get there, we must first define two helper functions.

Since the encoded terms have more subterms than their decoded versions due to the added i-symbols, we need a function that translates a position from the original to the encoded version. This will be needed when we have to find at what position the next redex in the constructed sequence is located.

The other function returns the position of the closest i-symbol above the input position. It will be used when an i-symbol needs to propagated inward.

Definition 25 (Position translation)**.**

Given some term t from the signature of an encoded TRS and a position π we define the following function:

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

Definition 26 (Next i above)**.**

Given some term t from the signature of an encoded TRS and a position π we define the following function:

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

Algorithm 31. (`CreateEncodedSequence`)

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

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

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

15:         if  $\pi_{k+1} \neq \omega$
16:             Let $\omega = \omega_1.\omega_2$ such that $\omega_2 \in \mathbb{N}$
17:             while  $root(s_{k,j}|_{\omega_1}) \neq \mathsf{i}$
18:                 $q \leftarrow \mathrm{n\_i}(s_{k,j}|_{\omega_1})$
19:                 $f \leftarrow root(s_{k,j}|_{q.1})$
20:                 $\beta_{k,j+1} \leftarrow \texttt{PROP}_f$
21:                 $\tau_{k,j+1} \leftarrow q$
22:                 $s_{k,j} \;\; = \;\; s_{k,j}[\texttt{PROP}^L\sigma]_{\tau_{k,j+1}}$
23:                 $s_{k,j+1} \leftarrow s_{k,j}[\texttt{PROP}^R\sigma]_{\tau_{k,j+1}}$
24:                 $\omega \leftarrow \mathrm{tr}(s_{k,j+1}, \pi_{k+1})$
25:                 Let $\omega = \omega_1.\omega_2$ such that $\omega_2 \in \mathbb{N}$
26:                 $j \leftarrow j + 1$
27:             end while

28:             $f \leftarrow root(s_{k,j}|_{\omega})$
29:             $\beta_{k,j+1} \leftarrow \texttt{EXE}_f$
30:             $\tau_{k,j+1} \leftarrow \omega_1$
31:             $s_{k,j} \;\; = \;\; s_{k,j}[\texttt{EXE}^L\sigma]_{\tau_{k,j+1}}$
32:             $s_{k,j+1} \leftarrow s_{k,j}[\texttt{EXE}^R\sigma]_{\tau_{k,j+1}}$
33:         end if

34:         $\alpha'_{k+1} \leftarrow \psi(\alpha_{k+1})$
35:         $\pi'_{k+1} \leftarrow \omega$
36:         $s_{k,j} \;\; = \;\; s_{k,j}[\varphi(\ell_{k+1})\sigma]_{\pi_{k+1}}$
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  $s_{0,0} \xrightarrow{0}_{\beta_{0,1},\tau_{0,1}} \cdots \xrightarrow{0}_{\beta_{0,m_1},\tau_{0,m_1}} s_{0,m_1} \to_{\alpha'_1,\pi'_1} s_{1,0} \to \cdots \to_{\alpha'_n,\pi'_n} s_{n,0}$
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 relative rules, but always ends by applying a single non-relative rule. In other words, the loop iterates over each rewrite step of the input rewrite sequence and 'simulates' it in the encoded TRS.

Let us now take a deeper look at each iteration. Lines 5 and 6 initialize a second counter j, which counts the relative rewrite steps, and a position ω, 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 bellow 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 relative rewriting, namely the rule and position respectively.

Line 8 chooses the position ω' 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 ω' and updates the value of j.

After the redex at ω has been made innermost, the procedure checks if ω equals the position π_{k+1} from the input rewrite sequence. If that is the case, then we can 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 bellow some i-symbol and is also not a redex yet. Therefore, the algorithm will propagate an i-symbol directly above the subterm at ω and restore its encoded symbol to the original defined symbol.

Line 16 splits ω in a way as to give us the position above it, namely ω_1. Then the while-loop at lines 17 to 27 checks after each iteration if the symbol at that position is i. The first two lines of the loop initialize a position q to the position of the closest i-symbol above ω_1, and then assigns f the symbol directly bellow the discovered i-symbol. More precisely that would be the root symbol at $q.1$. The next lines define the next relative rewrite step analogously to the one seen in the previous while-loop. Since propagation of i-symbols changes the structure of the term, the algorithm must update the value of ω_1 and consequently ω.

When the algorithm reaches line 28 after exiting the previous while-loop, the exact conditions to apply `EXE` at ω_1 are met and the defined symbol is restored at ω. The goal of the if-branch from lines 15 to 33 was to create a redex at position ω, where none was there prior to it. The final non-relative rule application is therefore shared by both cases. Line 38 increments k in preparation of 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 $\Phi(\mathcal{R})$.

Lemma 28 ($\mathrm{rc}_{\mathcal{R}} \subseteq \mathrm{irc}_{\mathcal{R}'}$)**.**

Let $\mathcal{R}$ be a TRS and $\Phi(\mathcal{R})$ its encoding. Let $t \in \mathcal{T}_{\mathcal{B}}(\Sigma)$ be some basic term.

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

Proof. We apply Lemma 26. to $t \to^n_{\mathcal{R}} v$ and obtain $\nabla = t \overset{oi}{\longrightarrow}{}^{n}_{\mathcal{R}} v'$ for some term v'. The procedure CreateEncodedSequence$(\mathcal{R}, \nabla)$ returns a rewrite sequence ∇' in $\Phi(\mathcal{R})$. Showing that the procedure terminates, that ∇' exists in $\Phi(\mathcal{R})$, that ∇' is innermost and of the same length, proves Lemma 28.

1. ∇' 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 $\omega = \pi_1 = \epsilon$ 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 lhs 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 lhs 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, its 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 bellow π_{k+1} not in NF are in $tag(t_k)$. The encoding Φ over-approximates these locations in $\mathcal{R}$ and subsequently it holds that in each rewrite sequence, subterms at positions in $tag(t_k)$ are encapsulated by i-symbols (1). It also holds that the root symbols of any subterm bellow these i-symbols is either another i-symbol or a constructor (2). Therefore, removing all i-symbols bellow the translated ω position, ensures that the subterm at it is not non-innermost. Each `OMIT` rewrite step is also innermost as per the definition of ω' at line 8. (3)

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 over-approximated to flow into a non-ndg location, i.e. the subterm at ω is encapsulated by an i-symbol.

Case I. ($\pi_{k+1} \neq \omega$) From (2) we also know that the root symbol at ω is a constructor, which by the definition of the translation function can be restored to some defined symbol. Since it is not necessary that the root symbol at ω_1, i.e. the position directly above ω, is an i-symbol, the procedure has to propagate one to it from the closest non-parallel position 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 ω. Each iteration calculates the new closest non-parallel position with root symbol i and propagates at it. These `PROP` rewrite steps are innermost, since all other i-symbols bellow ω have been removed and because it follows from (2) that the subterm at the position calculated by n_i is innermost (4).When this while-loop terminates, the `EXE` rewrite step restores the defined symbol at the now updated ω.

Case II. ($\pi_{k+1} = \omega$) In this case we know that the subterm at ω is already an innermost redex. For subterm that were not in NF 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 ω. We now have to show that the redex can be matched to ℓ_{k+1}. It follows from the procedure so far that $dec(s_{k,j}) = t_k$. 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 ω 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 $(3-4)$ cover 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 non-relative step per iteration is also innermost.

3. ∇' is has a length of n

The length of rewrite sequences is only affected by non-relative rewrite steps. The big while-loop in the procedure produces a single non-relative 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 bellow position ω 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 ω 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 2_

□

Definition 27 (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

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

Lemma 28.

Let $\mathcal{R}$ be a TRS and $\Phi(\mathcal{R}) = \mathcal{R}' \uplus \mathcal{S}$ its encoding, where $\mathcal{S}$ holds all the relative rules.

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

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 $dec(t_j) = dec(t_k), \forall j, k$. 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, $dec(v_j) = dec(v_k), \forall j, k$.

Therefore, proving $t_n \xrightarrow{i}_{\mathcal{R}'} v_0 \Rightarrow dec(t_n) \to dec(v_0)$ also shows $dec(t) \to dec(v)$.Let $t_n \to_{\psi(\alpha),\pi} v_0$. We define τ via $\pi = tr(t_n, \tau)$. Let $t_n = C[\varphi(\ell)\sigma]$. Define context $D = dec(C)$ with $D|_\tau = \square$. We have

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

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

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

This gives us the rewrite step

$$D[\ell\sigma'] = dec(t_n) \to_{\alpha,\tau} dec(v_0) = D[r\sigma']$$

and we are done.

□

Lemma 29. ($\mathrm{rc}_{\mathcal{R}} \supseteq \mathrm{irc}_{\Phi(\mathcal{R})}$)

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.

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

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

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

We apply Lemma 1. on the last $u \xrightarrow{i}_{\Phi(\mathcal{R})} v$ step to get $dec(u) \to_{\mathcal{R}} dec(v)$. We now have the sequence

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

and we are done.

□

Corollary 30. ($\mathrm{rc}_{\mathcal{R}} = \mathrm{irc}_{\Phi(\mathcal{R})}$)

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

5. Results

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.