Bachelor Thesis

Termgraph Rewriting

Maria Anna Schett
csag9385@uibk.ac.at

14 November 2011

Supervisors: Assoz.-Prof. Priv.-Doz. Dr. Georg Moser
MSc. Martin Avanzini

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Ich erkläre mich mit der Archivierung der vorliegenden Bachelor Thesis einverstanden.

Datum	Unterschrift

Abstract

This thesis provides an introduction to termgraph rewriting – being an adequate model to simulate term rewriting, i.e., a term s rewrites to a term t if for the corresponding termgraphs S, T it holds that S (graph) rewrites to T. This work focuses on outermost rewriting. Outermost rewriting is covered intently by exploring potential benefits introduced particularly by the strategy. In the course of this work outermost graph rewriting has been implemented in the functional programming language Haskell, providing ground work for an efficient implementation and serving laboratory purposes.

Acknowledgments

Special thanks to my supervisors Martin Avanzini and Georg Moser, who guided me through this project, especially to Martin, who even gave me his free time. Thanks to Julian Nagele, who supported me with insight and thanks to my family and friends, who didn't have a clue what I was up to.

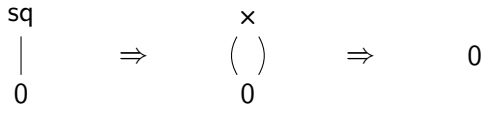
Contents

1 Introduction

Term rewriting is a Turing-complete, yet easy to understand, model of computation. Evolving from equational theory, term rewriting differs by introducing a direction to equations. This limitation of applicability of equations – i.e., only directed – yields a model of computation and raises the question of possible termination and confluence of a derivation. Term rewriting has its application in algebra, recursion theory, software engineering and programming languages [4]. Here especially functional programming languages ought to be mentioned, bearing great resemblance to term rewriting. Consider the function square (sq); deriving from mathematical knowledge one knows: $\mathsf{sq}(x) = x \times x$. Orientation of this equation from left to right leads to $\mathsf{sq}(x) \to x \times x$ as first example for a rewrite rule, and consider $x \times 0 \to 0$ as a second. Please note that term rewriting works on a syntactical level and the semantics are merely coincidental. Informally these rules can be applied to the term $\mathsf{sq}(0)$ leading to the following derivation:

$$\mathsf{sq}(0) \to 0 \times 0 \to 0$$

The general idea of term rewriting can be lifted to another data structure than terms – giving rise to graph rewriting. By representing terms as graphs, one gets an amiable feature, namely *sharing*. Thus please consider the above derivation on a graphical level.



One may observe that in the first step of this graphical derivation 0 was not duplicated as it happened in the above term rewrite sequence. This turns out to be the main benefit of graph rewriting. To fully benefit from this observation in all possible cases one has to establish that graph rewriting is sufficient to simulate term rewriting. This simulation is interesting because graph rewriting allows a precise control over the resources occupied. Thus the following theories have evolved.

In [8] and again in [2] it has been established that graph rewriting is adequate to simulate term rewriting. In [3] the simulation was restricted to innermost term rewriting and leading to a more efficient implementation. The aim of this thesis is to inspect adequacy for outermost graph rewriting with the outlook of describing an efficient implementation similar to [3].

The second part of this thesis consists of an implementation of outermost graph rewriting in a programming language of my choice. This part was extended throughout the thesis, finally providing a laboratory tool to experiment with graph rewriting and providing groundwork for further implementation.

This thesis is structured as follows. Chapter 2 covers the preliminaries: Term rewriting and graph rewriting. Chapter 3 shows the relationship between term rewriting and graph rewriting, i.e., that graph rewriting can be used to simulate term rewriting. This chapter focuses on different evaluation strategies for graph rewriting. The goal of implementing the outermost evaluation strategy is explored – following up the idea of [3], which focuses on innermost graph rewriting. A technique for implementing outermost rewriting successfully, as presented in [2] for full rewriting, is introduced. Chapter 4 presents the implementation of graph rewriting in Haskell by giving an introduction to the interface(s) of the tool and shortly presenting parts of the implementation. Chapter 5 deals with the evaluation of this aforementioned implementation. Therefore first a theoretical approach is taken. The advantages of graph rewriting are illuminated. Following up is a practical evaluation shedding new light on the program. The thesis concludes in Chapter 6 with gathered knowledge from this project.

2 Preliminaries

Let R be a binary relation on a set S and $x, y \in S$. Let R^n denote the *n-fold* composition of R, i.e., $xR^n y$ if there are elements $x_0 \dots x_n \in S$ such that $x = x_0 R x_1 \dots R x_n = y$. The transitive closure of R is denoted by R^+, i.e., xR^+y if $xR^n y$ for some $n \geq 1$. The transitive and reflexive closure is denoted by R^*, i.e., xR^*y if xR^+y or $x = y$. An element $x \in S$ is called *R-minimal* if there exists no element y such that xRy. Let $xR^!y$ denote xR^*y and y is *R-minimal*.

Let f be a function $f : A \to B$ and $C \subseteq A$. The operator $\restriction$ restricts a function to a subset of its domain. That is, $(f\restriction C) : C \to B$ with $(f\restriction C)(x) = f(x)$.

2.1 Term Rewriting

Following mainly [4, 6] the concepts and notions of term rewriting will be introduced. Throughout the chapter a real world example, namely addition and multiplication of natural numbers, is used to illuminate the theory.

2.1.1 Terms

First the notion of terms will be formally introduced. Thereafter terms will be inspected more closely by defining operations and relations on them.

Throughout this thesis, by $\mathcal{V}$ a countably infinite set of *variables* is denoted. Variables will usually be denoted by $x, y, z \dots$

Definition 2.1. Let $\mathcal{F}$ denote a non-empty, finite set of *function symbols*. Every function symbol f is associated with a natural number, which indicates its arity and is denoted by $\mathsf{ar}(f)$. A function symbol f with $\mathsf{ar}(f) = 0$ is called a constant. Let $\mathcal{F}$ be called a signature.

Definition 2.2. A *term* is constructed over a signature $\mathcal{F}$ and variables $\mathcal{V}$, where $\mathcal{F} \cap \mathcal{V} = \varnothing$. A term is defined inductively.

- Every $x \in \mathcal{V}$ is a term.
- If $t_1, \dots, t_n$ are terms and $f \in \mathcal{F}$, where $\mathsf{ar}(f) = n$, then $f(t_1, \dots, t_n)$ is a term.

Let $\mathcal{T}(\mathcal{F}, \mathcal{V})$ denote the (countably) infinite set of terms over $\mathcal{F}$ and $\mathcal{V}$. In $\mathcal{V}\mathsf{ar}(t)$ all variables occurring in t are collected.

Some conventions throughout this thesis will be settled now. Function symbols will be denoted by $f, g, h, \dots$ throughout this work, whereas constants will be treated differently by choosing from $a, b, c....$ Furthermore $s, t \dots$ will denote terms.

Example 2.3. Consider the signature $\mathcal{F} = \{\mathsf{0}, \mathsf{s}, +, \times\}$ to model addition and multiplication of natural numbers. The arities are $\mathsf{ar}(\mathsf{0}) = 0$, i.e., $\mathsf{0}$ is a constant, $\mathsf{ar}(\mathsf{s}) = 1$ and $\mathsf{ar}(+) = \mathsf{ar}(\times) = 2$. Then for instance $\mathsf{0}, \mathsf{s}(\mathsf{0}), +(\mathsf{0}, \mathsf{s}(\mathsf{0}))$ are terms. Usually infix notation is used when appropriate, i.e., $+(\mathsf{0}, \mathsf{s}(\mathsf{0}))$ is written as $\mathsf{0} + \mathsf{s}(\mathsf{0})$

A unique starting point for further inspecting terms is provided by the definition of the root of a term.

Definition 2.4. The *root* of a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$, $\mathsf{rt}(t)$, is defined as

$$\mathsf{rt}(t) := \begin{cases} x & \text{if } t \text{ is a variable } x \\ f & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

The inductive definition of terms implies the existence of subterms.

Definition 2.5. Let $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$. The set of *subterms* of t is defined by

$$\mathsf{subterms}(t) := \begin{cases} \{t\} & \text{if } t \text{ is a variable} \\ \{t\} \cup \mathsf{subterms}(t_1) \cup \cdots \cup \mathsf{subterms}(t_n) & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

If $s \in \mathsf{subterms}(t)$ this is also written as $s \trianglelefteq t$.

Example 2.6. The term $t = \mathsf{0} + (\mathsf{0} \times \mathsf{0})$ has the following subterms

$$\{\mathsf{0} + (\mathsf{0} \times \mathsf{0}), \mathsf{0}, \mathsf{0} \times \mathsf{0}\},$$

hence for instance $\mathsf{0} \trianglelefteq t$.

Observing that the term $\mathsf{0}$ occurs thrice in t in the above example, this leads to the question how to uniquely address a subterm in t. The concept of positions will be introduced to tackle this problem.

Definition 2.7. A *position* is a finite sequence of natural numbers. The empty position is denoted by ϵ. Position p and q may be concatenated by $\cdot$, i.e., $p \cdot q$, where position $p \cdot q = p$ if $q = \epsilon$.

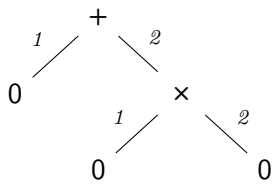
Definition 2.8. Let p be a position and t be a term, then the subterm of t at position p, $t|_p$, is defined as

$$t|_p := \begin{cases} t & \text{if } p = \epsilon \\ t_i|_q & \text{if } p = i \cdot q \text{ and } t = f(t_1, \ldots, t_i, \ldots, t_n) \end{cases}$$

All positions of s in t are collected in $\mathsf{pos}(s, t) := \{p \mid t|_p = s\}$.

To clarify the definition of position it is useful to represent a term in tree-form.

Example 2.9. Consider the term $t = \mathsf{0} + (\mathsf{0} \times \mathsf{0})$. Then t is represented as the following labeled tree (with labels over $\mathcal{F} \cup \mathcal{V}$), where positions are indicated as edge-labels.



From this tree one can derive the following sets: $\mathsf{pos}(0+(0\times 0),t) = \{\epsilon\}$, $\mathsf{pos}((0\times 0),t) = \{2\}$ and $\mathsf{pos}(0,t) = \{1, 2\cdot 1, 2\cdot 2\}$.

Kindly note the correspondence between positions and paths in the tree representation, a fact that is exploited later on. The tree representation also suggests some relations between positions.

Definition 2.10. A position p is *above* a position p' if p is a prefix of p', i.e., $pq = p'$ for some position q. A position p is *below* a position p', if p' is above position p, i.e., $p'q = p$ for some q. Two positions are *parallel* if neither is above or below the other.

2.1.2 Term Rewriting

So far terms and operations and relations on terms have been introduced, but no way to manipulate terms was indicated. So at this point one can only write a syntactically correct term down – whereas in this section a way to re-write them shall be explored.

Definition 2.11. A *rewrite rule* $l \to r$ consists of $l, r \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ satisfying the constraints:

- $\mathsf{rt}(l) \notin \mathcal{V}$.
- $\mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(l)$

A *term rewrite system* (TRS for short) is a set of rewrite rules.

Throughout this thesis $\mathcal{R}$ will denote a term rewrite system. To apply such a rewrite rule to a term s, one has to find a pattern between s and the left hand side of this rule. This is done by substituting variables.

Definition 2.12. A *substitution* is a mapping $\sigma\colon \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$, such that for finitely many x holds $\sigma(x) \neq x$. A renaming is a bijective substitution with $\mathcal{V} \to \mathcal{V}$.

Definition 2.13. Let t be a term and σ some substitution. The extension of σ to terms $\hat{\sigma}\colon \mathcal{T}(\mathcal{F}, \mathcal{V}) \to \mathcal{T}(\mathcal{F}, \mathcal{V})$, i.e., an endomorphism, is defined as

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

Usually one does not distinguish between the substitution σ and its extension $\hat{\sigma}$. Following convention $\hat{\sigma}(t)$ will be denoted by $t\sigma$. A term t' is called a variant of t if there exists a renaming σ_r such that $t\sigma_r = t'$.

Example 2.14. Consider the following substitution $\sigma = \{x \mapsto 0, y \mapsto 0 \times 0\}$ and the term $t = x + y$. The application of σ leads to $t\sigma = 0 + (0 \times 0)$. To illustrate a renaming consider $\sigma_r = \{x \mapsto y, y \mapsto z\}$. When applied to t with $t\sigma_r = y + z$ one gets a variant of t.

When replacing within a term some subterm with another term, a context is necessary.

Definition 2.15. Extend $\mathcal{F}$ by a fresh constant symbol $\Box$, called the hole. A term $c \in \mathcal{T}(\mathcal{F} \cup \{\Box\}, \mathcal{V})$ is called a *context* if $\Box$ occurs exactly once in c, i.e., $\mathsf{pos}(\Box, c)$ is a singleton.

Example 2.16. Consider the context $c = 0 \times \Box$ and the empty context $c_e = \Box$ exemplary.

One can think of a context as a term with a hole, which can be made whole by filling the hole with an arbitrary term. This does not exactly match the definition, as it degrades contexts to incomplete terms, which they are not. Nevertheless this image may clarify the intuition behind contexts.

Definition 2.17. A context c is denoted by $C[\Box]_p$ if $c|_p = \Box$. $C[t]_p$ denotes the replacement of $\Box$ by the term t.

Example 2.18. Reconsider the context $c = 0 \times \Box$ from above and the term $t = 0 \times 0$, then $C[\Box]_2$ and $C[t]_2 = 0 + (0 \times 0)$. When "filling" the empty context $c_e = \Box$ where $C[\Box]_\epsilon$ with t one gets $C[t]_\epsilon = t = 0 \times 0$.

This concludes the list of ingredients to formally define a rewrite step.

Definition 2.19. Let $\mathcal{R}$ be a TRS. A term s rewrites to a term t, denoted by $s \to_\mathcal{R} t$ if there exists a rule $l \to r \in \mathcal{R}$, a substitution σ and a context $C[\Box]_p$ such that $s = C[l\sigma]_p$ and $t = C[r\sigma]_p$.

When special emphasis on the position at which the rewrite step was performed is demanded, this will be indicated by $\to_{\mathcal{R},p}$.

Example 2.20. The term

$$s = 0 \times (0 + \mathsf{s}(0))$$

rewrites with the rewrite system

$$\mathcal{R} = \{0 \times x \to x, 0 + x \to x\}$$

using the rule

$$l \to r = 0 + x \to x$$

with substitution

$$\sigma = \{x \mapsto \mathsf{s}(0)\}$$

and context

$$C[\Box]_2 = 0 \times \Box.$$

The application of σ to l, r leads to

$$l\sigma = \mathsf{0} + \mathsf{s}(\mathsf{0}) \text{ and } r\sigma = \mathsf{s}(\mathsf{0}).$$

By replacing the hole one gets

$$C[l\sigma]_{\mathcal{Q}} = \mathsf{0} \times (\mathsf{0} + \mathsf{s}(\mathsf{0})) = s.$$

So by

$$C[r\sigma]_{\mathcal{Q}} = \mathsf{0} \times (\mathsf{s}(\mathsf{0})) = t$$

one concludes $s \to_{\mathcal{R}} t$.

To sum up this section: A term s can be rewritten to a term t, whenever there is a rule $l \to_{\mathcal{R}} r$ applicable to s, i.e., a substitution from l to s at some position p in s can be found. The rewrite step is then concluded by applying this substitution on r and placing this instantiated r at position p in s.

2.1.3 Rewrite Sequences

Having managed the first step, this section deals with questions arising when rewrite steps are applied repeatedly to a term.

A term s is called *reducible* with respect to a TRS $\mathcal{R}$ if there exists a term t such that $s \to_{\mathcal{R},p} t$ at some position p. The subterm $s|_p$ is then called a *reducible expression*, for short *redex*. The plural of redex is hereby defined as redices.

A computation halts if there are no more reducible expressions, i.e., if no rewrite step is possible anymore.

Definition 2.21. A term t is in normal form if it is not reducible. The set of all normal forms corresponding to a TRS $\mathcal{R}$ is denoted by $\mathsf{NF}(\mathcal{R})$. Let $s \to^{!}_{\mathcal{R}} t$ denote $s \to^{*}_{\mathcal{R}} t$ and $t \in \mathsf{NF}(\mathcal{R})$.

Example 2.22. Recall the TRS $\mathcal{R} = \{\mathsf{0} \times x \to \mathsf{0}, \mathsf{0} + x \to x\}$. In Example 2.20 the step

$$\mathsf{0} \times (\underline{\mathsf{0} + \mathsf{s}(\mathsf{0})}) \to_{\mathcal{R}} \mathsf{0} \times (\mathsf{s}(\mathsf{0}))$$

was performed. The redex in this example is $\underline{\mathsf{0} + \mathsf{s}(\mathsf{0})}$, indicated through underline. Another redex can be found in this result and thus

$$\underline{\mathsf{0} \times (\mathsf{s}(\mathsf{0}))} \to_{\mathcal{R}} \mathsf{0}$$

No more rule is applicable on the term $\mathsf{0}$ and the computation halts, hence $\mathsf{0}$ is in normal form.

Considering again the start term $s = \mathsf{0} \times (\mathsf{0} + \mathsf{s}(\mathsf{0}))$ one might argue, that this start term contains another redex, giving rise to the rewrite step

$$\underline{\mathsf{0} \times (\mathsf{0} + \mathsf{s}(\mathsf{0}))} \to_{\mathcal{R}} \mathsf{0}$$

A term may contain several reducible expressions. An evaluation strategy provides a mechanism to uniquely choose a redex out of several such. Therefore positions will be considered again.

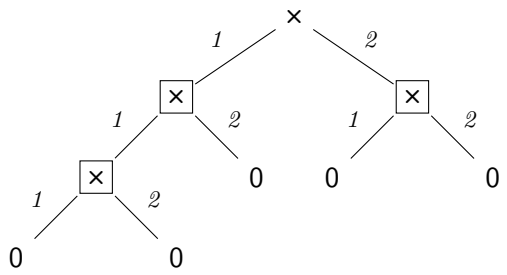
Definition 2.23. A redex t within term s is outermost (innermost) if $s|_p = t$ and there is no other redex $t' = s|_{p'}$ such that p' is above (below) p.

By $s \xrightarrow{\text{o}}_{\mathcal{R}} t$ ($s \xrightarrow{\text{i}}_{\mathcal{R}} t$) the outermost (innermost) rewrite step $s \to_{\mathcal{R}} t$ is meant.

Example 2.24. Consider $\mathcal{R} = \{x \times 0 \to 0\}$ and the term

$$s = ((0 \times 0) \times 0) \times (0 \times 0)$$

represented as a tree with the redices marked through boxes as follows.



Then $s|_{1}$ is outermost as position *1* is above $s|_{1\,.\,1}$ which is an innermost redex. The redex $s|_{2}$ is innermost as much as outermost.

This example shows, that there might still be multiple outermost (innermost) redices.

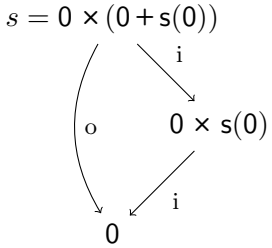
Definition 2.25. A redex position p is *leftmost* outermost (innermost) if for all parallel outermost (innermost) redex positions q, $p <_{lex} q$ holds.

Similarly rightmost redices are defined, but throughout this thesis only leftmost outer- or innermost evaluation will be employed. It is implicitly assumed, when just innermost or outermost is mentioned.

Note that although a rewrite strategy is employed, there might still be different outcomings of a computation. This is due to the fact, that multiple rules may apply to the same redex and no order of choosing rules is defined. Still producing a unique result is a desired property of a rewrite system.

Definition 2.26. A term $s \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ is called *confluent* wrt. to a TRS $\mathcal{R}$ if whenever $s \to^*_{\mathcal{R}} t_1$ and $s \to^*_{\mathcal{R}} t_2$ there exists a term t such that $t_1 \to^*_{\mathcal{R}} t$ and $t_2 \to^*_{\mathcal{R}} t$. A TRS $\mathcal{R}$ is confluent if all terms are confluent with respect to $\mathcal{R}$.

Example 2.27. Consider the term s and the two possible rewrite sequences shown below.



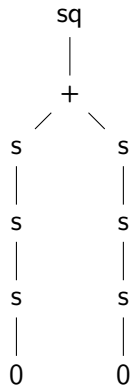
Thus s is confluent with respect to $\mathcal{R}$.

2.2 Graph Rewriting

Ever so often a different representation of data allows some finesse. Therefore now a new representation of terms will be discussed. As a motivating example consider the term

$$t = \mathsf{sq}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0))) + \mathsf{s}(\mathsf{s}(\mathsf{s}(0))))$$

As already introduced in Example 2.9, a term might be represented as a tree. This representation of t will be depicted in the following figure.

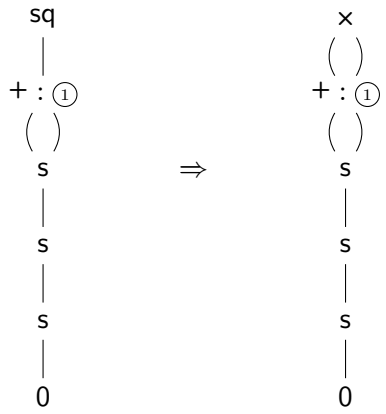


Note that the subterm $\mathsf{s}(\mathsf{s}(\mathsf{s}(0)))$ occurs twice in t. This suggests a more space-efficient representation as a graph. Or more precise – as a directed acyclic graph. The next figure illustrates this.

So far this sharing of nodes saves space (in memory at least) but a more interesting feature comes up, when applying a rewrite step. Therefore consider the rule $\mathsf{sq}(x) \to x \times x$. With term rewriting the derivation is

$$t = \mathsf{sq}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0))) + \mathsf{s}(\mathsf{s}(\mathsf{s}(0)))) \to (\mathsf{s}(\mathsf{s}(\mathsf{s}(0))) + \mathsf{s}(\mathsf{s}(\mathsf{s}(0)))) \times (\mathsf{s}(\mathsf{s}(\mathsf{s}(0))) + \mathsf{s}(\mathsf{s}(\mathsf{s}(0))))$$

This step can be simulated with termgraphs, which are specified graphs corresponding to a term. Eager ones may forward to Definition 2.31. Intuitively a termgraph corresponding to the above mentioned term t and the corresponding rewrite step would look like this:



Note that copying the argument of square was avoided by having two edges point to same subgraph starting from ①. So where the term's size doubled through the rewrite step, the growth of the termgraph was fixed. When not a single but multiple steps are performed, the term's blow-up might be exponential in the length of the rewrite sequence. By not duplicating subterms this does not happen with graph rewriting.

After this informal introduction to termgraphs the remainder of this chapter introduces termgraphs formally. Further on it illustrates and defines graph rewriting.

2.2.1 Term Graphs

First an appropriate graph structure will be discussed, followed up by the definition of termgraph. Furthermore the relation between terms and termgraphs will be shown. Afterwards the novelty introduced by termgraphs – the possibility of sharing nodes – will be discussed. As a way to distinguish subterms representing the same subtermgraph positions are re-used. The notations follow mainly [2, 3].

A termgraph is based on a graph. Therefore first a structure for graphs will be introduced.

Let $\mathsf{G} = (\mathsf{V_G}, \mathsf{Succ_G}, \mathsf{L_G})$ be a directed graph over a set of labels $\mathcal{L}$, such that

- $\mathsf{V_G}$ denotes a set of nodes or vertices.
- $\mathsf{Succ_G}$ is a mapping $\mathsf{V_G} \to [\mathsf{V_G}, \ldots, \mathsf{V_G}]$, every node maps to an ordered list of successors.
- $\mathsf{L_G}$ is a mapping $\mathsf{V_G} \to \mathcal{L}$, connecting every node to a label.

To indicate that a node n is part of G, i.e., $n \in \mathsf{V_G}$, in the following this will be abbreviated to $n \in \mathsf{G}$.

Navigation through a directed graph G is done via the successor relation. Suppose for some node $n \in \mathsf{G}$ holds $\mathsf{Succ_G}(n) = [m_1, \ldots, m_i, \ldots, m_k]$. Then m_i is the ith successor of n denoted by $n \overset{i}{\rightharpoonup} m_i$. If i is of no importance this is simply denoted by $n \rightharpoonup m$. Recall that $\rightharpoonup^*$ denotes the reflexive and transitive closure, as well as $\rightharpoonup^+$ denotes the transitive closure of $\rightharpoonup$.

Definition 2.28. Let G be a graph and V' denote the set of vertices reachable from a node $n \in \mathsf{G}$, i.e., $\mathsf{V}' := \{m \mid n \rightharpoonup^* m\}$. Recall the operator $\restriction$ from the beginning. Let $\mathsf{G}' := (\mathsf{V}', \mathsf{Succ}\restriction\mathsf{V}', \mathsf{L}\restriction\mathsf{V}')$. Then G' is the *subgraph* of G reachable from n. This will be denoted by $\mathsf{G}\restriction n$, overloading the operator $\restriction$.

Further consider the definition of a directed, acyclic and rooted graph.

Definition 2.29. Let G be a directed graph as introduced above. Let DAG be such a graph where additionally holds:

- G is rooted, i.e., there exists a unique root node $\mathsf{rt}(\mathsf{G})$ such that for all nodes n, $\mathsf{rt}(\mathsf{G}) \rightharpoonup^* n$ holds.
- G is acyclic, i.e., $n \rightharpoonup^+ m$ implies $n \neq m$ for all nodes n, m in G.

A node in a graph DAG can be shared.

Definition 2.30. A node $n \in \mathsf{DAG}$ is *shared* if there exist at least two distinct paths $\mathsf{rt}(\mathsf{DAG}) \overset{i_1}{\rightharpoonup} \cdots \overset{i_k}{\rightharpoonup} n$ and $\mathsf{rt}(\mathsf{DAG}) \overset{j_1}{\rightharpoonup} \cdots \overset{j_l}{\rightharpoonup} n$ i.e., there is an index z such that $i_z \neq j_z$ with $z \leq k$ and $z \leq l$.

For such a graph DAG to be sufficient to model a termgraph, some restrictions ought to hold.

Definition 2.31. Let DAG be a directed, acyclic and rooted graph. Then DAG is a *termgraph* when

- the set of labels $\mathcal{L}$ is $\mathcal{F} \cup \mathcal{V}$.
- for a node $n \in \mathsf{DAG}$ with $\mathsf{L}_{\mathsf{DAG}}(n) = f \in \mathcal{F}$ and $\mathsf{ar}(f) = k$ it holds that $\mathsf{Succ}_{\mathsf{DAG}}(n) = [n_1, \ldots, n_k]$
- variable nodes do not have successors, i.e., for every $n \in \mathsf{DAG}$ with $\mathsf{L}_{\mathsf{DAG}}(n) \in \mathcal{V}$ holds $\mathsf{Succ}_{\mathsf{DAG}}(n) = []$.
- for all $n_1 \in \mathsf{DAG}$ with $\mathsf{L}_{\mathsf{DAG}}(n_1) = x \in \mathcal{V}$ holds if $\mathsf{L}_{\mathsf{DAG}}(n_2) = x$ then $n_1 = n_2$.

Let S be such a termgraph, then $\mathcal{V}\mathsf{ar}(S) := \{n \mid \mathsf{L}_s(n) \in \mathcal{V}\}$ denotes the set of all nodes representing a variable in S.

To transform a termgraph into the term it represents, consider the following definition.

Definition 2.32. The term representation of a termgraph S is defined by

$$\mathsf{term}(S) := \begin{cases} x & \text{if } \mathsf{L}(\mathsf{rt}(S)) = x \in \mathcal{V} \\ f(\mathsf{term}(S\restriction n_1), \ldots, \mathsf{term}(S\restriction n_k)) & \text{if } \mathsf{L}(\mathsf{rt}(S)) = f \in \mathcal{F} \\ & \text{and } \mathsf{Succ}(n) = [n_1, \ldots, n_k] \end{cases}$$

In illustrated termgraphs a node n with $\mathsf{L}(n) = l$ will be depicted as

$$l : \textcircled{n}$$

Possibly n will be omitted if only the label is of importance.

Example 2.33. The termgraph representation of $x + x$ follows

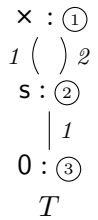
As required by definition the variable node ② is shared.

The question of how to uniquely address a subterm arises again. This becomes even more tricky now, as due to sharing, subterms could really be represented by the same structure. The answer stays the same: positions.

Definition 2.34. Let S be a termgraph and n be a node in S. The set of positions of n in S is defined by $\mathsf{pos}(S, n) := \{i_1 \cdot \dots \cdot i_k \mid \mathsf{rt}(S) \xrightarrow{i_1} \cdots \xrightarrow{i_k} n\}$. In $\mathsf{pos}(S) := \bigcup_{n \in S} \mathsf{pos}(S, n)$ all positions in S are collected. Vice versa, for position $p = i_1 \cdot \dots \cdot i_k \in \mathsf{pos}(S)$ the node n corresponding to this position is defined as $\mathsf{node}(S, p) := n$ where $\mathsf{rt}(S) \xrightarrow{i_1} \cdots \xrightarrow{i_k} n$.

To clarify positions within a termgraph consider the following example.

Example 2.35. Here the successor position is explicitly indicated by edge labels.



One can see that node ③ is reachable by multiple paths. So the set is

$$\mathsf{pos}(T, ③) = \{1 \cdot 1, 2 \cdot 1\}$$

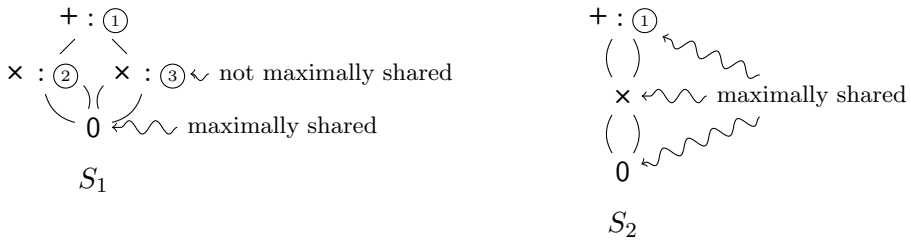
Accordingly the node at those positions is

$$\mathsf{node}(T, 1 \cdot 1) = \mathsf{node}(T, 2 \cdot 1) = ③$$

Note that the size of $\mathsf{pos}(S, n)$ corresponds to the amount of subterms this node represents. This provides another way to detect whether a node n is shared. Clearly n in S is shared whenever $|\mathsf{pos}(S, n)| > 1$.

Definition 2.36. A node $n \in S$ is *maximally shared* if whenever $\mathsf{term}(S{\restriction}n) = \mathsf{term}(S{\restriction}m)$ then $n = m$. So the same subterm is represented by the same node. A termgraph S is *maximally sharing* if all its nodes are maximally shared. A node is *minimally shared* if it is either unshared or a variable node. A termgraph S is *minimally sharing* if all its nodes are minimally shared.

Example 2.37. Consider the following termgraphs S_1, S_2 both representing the term $(0 \times 0) + (0 \times 0)$.



The graph S_1 is not maximally sharing because not all its nodes are maximally sharing, i.e., ③ is not maximally shared as $\mathsf{term}(S_1|\textcircled{2}) = \mathsf{term}(S_1|\textcircled{3})$ but $\textcircled{2} \neq \textcircled{3}$. The corresponding maximally sharing termgraph is depicted as S_2. Note here especially that the root, ① is maximally *and* minimally sharing.

This concludes the introduction of termgraphs, now a way to rewrite them will be explored.

2.2.2 Rewriting - A Graphical Approach

In this section the mechanism behind graph rewriting will be illustrated and defined. Therefore first an overview over the necessary (intermediate) steps will be given. Intuitively, applying a graph rewrite rule $L \Rightarrow R$ to a redex node n in a term graph S amounts to the following steps:

1. Determine the graph morphism m between the left hand side L and the subgraph rooted at n in S. The graph morphism plays the role of the substitution in term rewriting.
2. Add a fresh copy of the right hand side R to S.
3. Apply the morphism m to the added right hand side by redirecting all variable nodes to the corresponding nodes in S.
4. Redirect all edges going to n to the root of R.
5. Finally remove all nodes that became inaccessible.

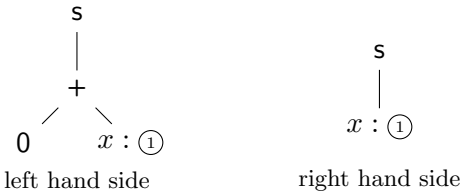
The next example serves the purpose to illustrate the definitions beforehand. This example will not illustrate the benefits of graph rewriting – but neither will it feature the problems to come.

Example 2.38. Consider the rule $\mathsf{s}(\mathsf{0}+x) \to \mathsf{s}(x)$ and the term $\mathsf{s}(\mathsf{s}(\mathsf{0}+\mathsf{s}(\mathsf{0})))$. Hence

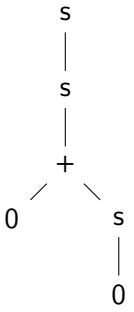
$$s = \mathsf{s}(\mathsf{s}(\mathsf{0}+\mathsf{s}(\mathsf{0}))) \to \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{0}))) = t \text{ with } \sigma = \{x \mapsto \mathsf{s}(\mathsf{0})\}$$

Now this step will be performed with graph rewriting following the above structure.

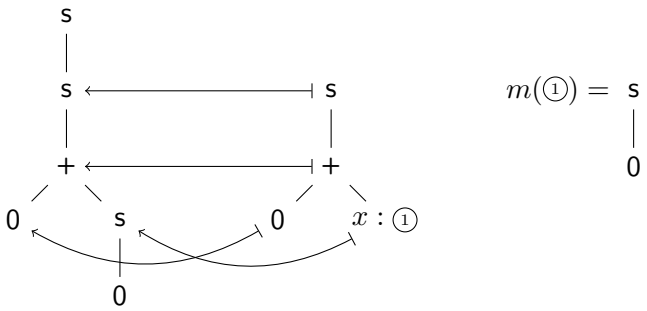
To simulate the rewrite step also the left hand side and the right hand side of the rule in $\mathcal{R}$ will be represented as termgraphs. Note that the variable node x is shared. This is indicated by the same node number ①.



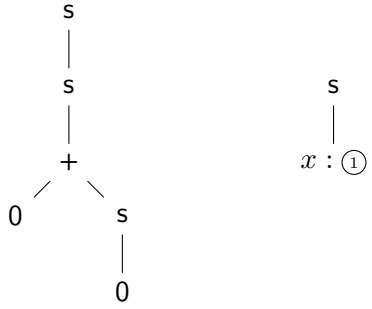
The term s will be transformed into a termgraph S.



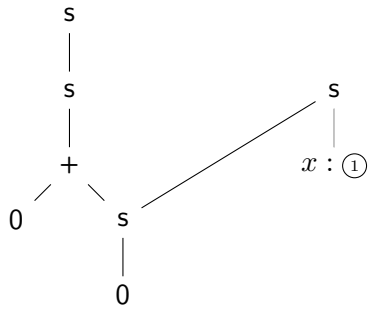
1. Now one finds a mapping (which plays the role of substitutions in term rewriting) between the left hand side of the rule and S. Therefore looking at the structure of the left hand side within S is required. Variables may be mapped to everything, this being the nature of variables.



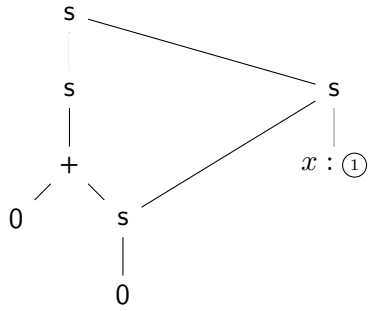
2 . Like in term rewriting the rewritten subgraph will be replaced by the right hand side of the rule. Recall the rewrite rule $\mathsf{s}(\mathsf{0}+x) \to \mathsf{s}(x)$ and the substitution $\sigma = \{x \mapsto \mathsf{s}(\mathsf{0})\}$. Adding a fresh copy of the right hand side of the rule to S leads to:



3. Applying the mapping from above (the "graph substitution") on the right hand side of the rule yields the following graph.



4. Now the right hand side of the rule is plugged into S instead of the rewritten subgraph. This is again done by redirecting edges, this time onto the root of the right hand side.



5. Finally one has to delete inaccessible nodes, i.e., only nodes reachable from the root are kept. This garbage collection corresponds to calling the subgraph ($\restriction$), starting from the root. In a picture:

This concludes the informal example graph rewrite step.

First the concept of redirecting a node will be defined. To avoid inconsistencies when merging two termgraphs consider the following definition.

Definition 2.39. Let S, T be termgraphs. S and T are *properly sharing*, if $n \in \mathsf{V}_S \cap \mathsf{V}_T$ implies $\mathsf{L}_S(n) = \mathsf{L}_T(n)$ and $\mathsf{Succ}_S(n) = \mathsf{Succ}_T(n)$. If S, T are properly sharing let $S \cup T$ denote the union of S and T, where

$$S \cup T := (\mathsf{V}_S \cup \mathsf{V}_T, \mathsf{Succ}_S \cup \mathsf{Succ}_T, \mathsf{L}_S \cup \mathsf{L}_T)$$

Note that the union is well defined because S and T are properly sharing.

Definition 2.40. Let S, T be properly sharing termgraphs and let n be a node with $n \in S$, but $n \notin T$. Let $r : \mathsf{V}_{S \cup T} \to \mathsf{V}_{S \cup T}$ be a function which replaces n by $\mathsf{rt}(T)$.

$$r(m) := \begin{cases} \mathsf{rt}(T) & \text{if } m = n \\ m & \text{otherwise} \end{cases}$$

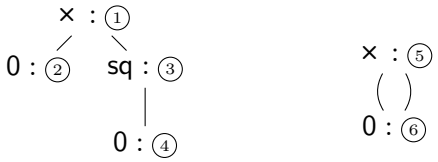
Informally, by applying r to $\mathsf{Succ}_{S \cup T}$ all edges pointing to n are redirected to $\mathsf{rt}(T)$. Let $(S \cup T)' = (V_{S \cup T}, \mathsf{Succ}_{(S \cup T)'}, \mathsf{L}_{S \cup T})$, where for all nodes m in $V_{S \cup T}$ if $\mathsf{Succ}_{S \cup T}(m) = [m_1, \ldots, m_k]$ then $\mathsf{Succ}_{(S \cup T)'}(m) = [r(m_1), \ldots, r(m_k)]$. Finally the redirection of n to the termgraph T, denoted by $S[T]_n$ is defined by

$$S[T]_n := \begin{cases} T & \text{if } n = \mathsf{rt}(S) \\ (S \cup T)' \restriction \mathsf{rt}(S) & \text{otherwise} \end{cases}$$

The second case deletes all nodes which are reachable only from n and thus are now inaccessible.

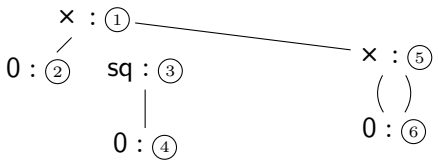
To clarify this definition now a stepwise example will be given.

Example 2.41. Consider the termgraphs S and T.



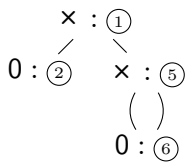
S $\qquad$ T

Note that $V_s \cap V_T = \varnothing$, hence they are properly sharing. Now consider the redirection of node ③ to T, i.e., $r(③) = ⑤$ and $r(ⓝ) = ⓝ$ for all other nodes in $S \cup T$.



$(S \cup T)'$

Finally $S[T]_③ = (S \cup T)' \restriction ①$ is depicted in the following.



$S[T]_③$

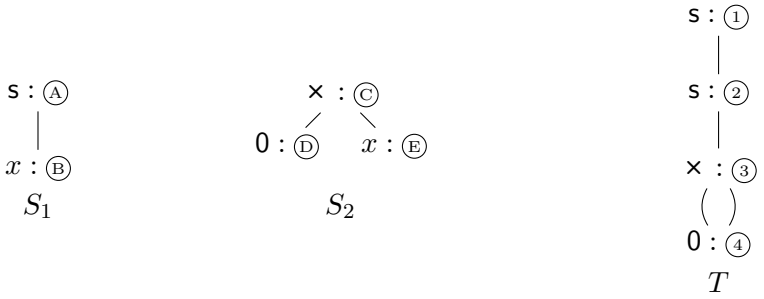
Recall Definition 2.13 where substitution was extended to an endomorphism on terms. Similarly a graph morphism poses an endomorphism on termgraphs.

Definition 2.42. A *morphism* between two termgraphs S, T is a mapping $m: S \to T$ such that

- $m(\mathsf{rt}(S)) = \mathsf{rt}(T)$
- for all $n \in S$ with $\mathsf{L}_S(n) \in \mathcal{F}$
  1. $\mathsf{L}_S(n) = \mathsf{L}_T(m(n))$
  2. if $\mathsf{Succ}_S(n) = [n_1, \ldots n_k]$ and $\mathsf{Succ}_T(m(n)) = [n'_1, \ldots n'_k]$ then

$$m(n_1) = n'_1, \ldots, m(n_k) = n'_k$$

Example 2.43. Consider the termgraphs S_1, S_2 and T.



Here it is possible to find the following morphism:

$$m_1: S_1 \to T \text{ with } Ⓐ \mapsto ①,\ Ⓑ \mapsto ②$$

Also one can find the morphism:

$$m_2 : S_1 \to T\restriction② \text{ where } Ⓐ \mapsto ②,\ Ⓑ \mapsto ③$$

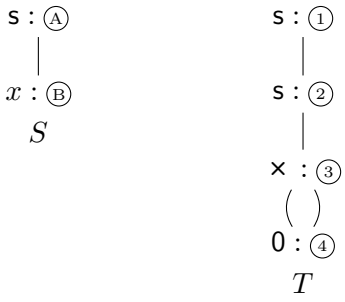
Morphisms being such a crucial concept deserve another example. Consider the termgraphs S_2 and T with

$$m_3 : S_2 \to T\restriction③ \text{ where } Ⓒ \mapsto ③,\ Ⓓ \mapsto ④,\ Ⓔ \mapsto ④$$

Definition 2.44. Let S, T be properly sharing termgraphs and $m : S \to T$ be a morphism. The application of the morphism m to S, denoted as $m(S)$, is defined by redirecting all variable nodes in S to their image. That is, for $\{n_1 \ldots n_k\} = \mathcal{V}\mathsf{ar}(S)$ set $S_0 := S$, $S_i := S_{i-1}[T\restriction m(n_i)]_{n_i}$ for $1 \leq i \leq k$ and $m(S) := S_k$. Or, in other words,

$$m(S) = (((S[T\restriction m(n_1)]_{n_1})[T\restriction m(n_2)]_{n_2}) \cdots)[T\restriction m(n_k)]_{n_k}$$

Example 2.45. Consider the termgraphs S and T



and the morphism $m : S \to T\restriction_{②}$ with Ⓐ $\mapsto$ ②, Ⓑ $\mapsto$ ③. The application $m(S)$ yields

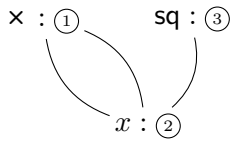
Definition 2.46. A *graph rewrite rule* consists of two properly and minimally sharing termgraphs L and R. Analog to term rewrite rules the following conditions must hold:

- $\mathsf{L}_L(\mathsf{rt}(L)) \notin \mathcal{V}$
- $\mathcal{V}\mathsf{ar}(R) \subseteq \mathcal{V}\mathsf{ar}(L)$

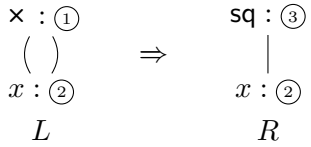
A graph rewrite rule is denoted by $L \Rightarrow R$. A set of graph rewrite rules is called *graph rewrite system* (GRS for short).

Recall the definition of minimally sharing and note that a variable node n is minimally shared. Also note that all variable nodes occurring in R also have to occur in L. Next are two ways to depict a graph rewrite rule.

Example 2.47. To emphasis properly sharing both rules could be depicted in a DAG. Note that hereby the root nodes have to be given, to be able to distinguish the left hand side from the right hand side. So $\mathsf{rt}(L) =$ ① and $\mathsf{rt}(R) =$ ③.



However, to be closer to the representation indicated by term rewrite rules, another possible representation is following.



Throughout this thesis the sharing of variable nodes will be indicated only by common node numbers.

For convenience the rules employ minimal sharing, because then, if no morphism can be found it is not because of inconvenient sharing of the graph rewrite rule.

As an invariant of termgraphs every node has to appear just once. Thus before combining two termgraphs S and T it has to be ascertained, that this fact holds up – by ascertaining that S and T are properly sharing. By making sure that the V_S and V_T are disjoint this condition can by met. This makes renaming a necessary technicality.

Definition 2.48. A *renaming* of a graph rewrite rule $L \Rightarrow R$ with respect to a termgraph S is the application of a bijective morphism $m : \mathsf{V}_{L \cup R} \to \mathsf{V}_{(L \cup R)'}$ such that $\mathsf{V}_S \cap \mathsf{V}_{(L \cup R)'} = \varnothing$.

Example 2.49. Recall the graph rewrite rule from Example 2.47. Further consider a termgraph S with the set of vertices $\mathsf{V}_S = \{①, ②\}$. This requires, for instance, the following renaming: $m : ① \mapsto ④,\ ② \mapsto ⑤$. The resulting rule $L' \Rightarrow R'$ is depicted below:

$$L' = \begin{array}{c} \times : ④ \\ (\ ) \\ x : ⑤ \end{array} \qquad \Rightarrow \qquad R' = \begin{array}{c} \mathsf{sq} : ③ \\ | \\ x : ⑤ \end{array}$$

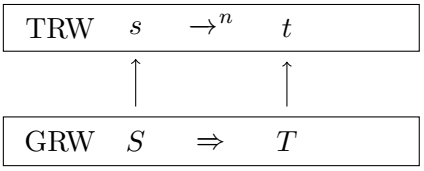
This concludes the necessary ingredients to define a graph rewrite step.

Definition 2.50. Given a graph rewrite system $\mathcal{G}$, a termgraph S *rewrites* to a termgraph T, denoted by $S \Rightarrow_{\mathcal{G}} T$, if there exists a graph rewrite rule $L \Rightarrow R \in \mathcal{G}$, where $L' \Rightarrow R'$ is a renaming of $L \Rightarrow R$ with respect to S, and a morphism $m : L' \to S\lceil n$ such that $S[m(R')]_n = T$.

3 Adequacy of Graph Rewriting

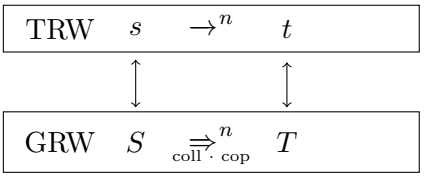
This chapter will establish a connection between the concepts of term rewriting and graph rewriting.

Given two models of computation, term rewriting and graph rewriting, which are so closely connected the question, arises: Can either be simulated by the other? In [8] soundness of termgraph rewriting with respect to term rewriting is shown.

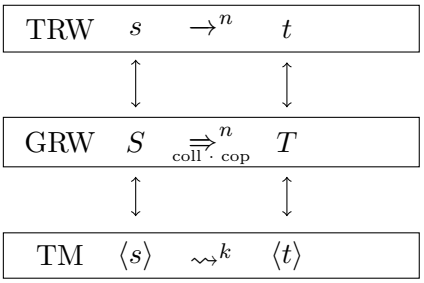


Every graph rewrite step can be simulated by n term rewrite steps, where n equals the amount of paths to the node, which represented the redex. So far this simulation works in one direction only and is interested in the results of the computation. The intermediate steps may differ.

The next step in [7] was to prove stepwise equality. To this end sharing (collapsing) and unsharing (copying)[1] of nodes were developed, which now allow a bi-directional simulation.



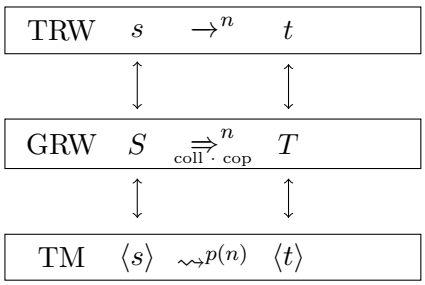
Hence graph rewriting together with sharing and unsharing of nodes is *adequate* to simulate term rewriting and vice verse. As mentioned in the introduction term rewriting is Turing-complete. Hence graph rewriting also is, as it can simulate term rewriting.



[1] Informally the sharing operation collapses two nodes, which represent the same subterm, whereas unsharing copies a shared node to obtain an unshared node representing the same subterm.

Here $\langle s \rangle$ denotes a suitable encoding of s on a Turing Machine (TM) and $\rightsquigarrow$ denotes the next configuration relation of a TM. The above picture indicates that functions computed by a TM can also be computed by term rewriting and consequently graph rewriting. However, so far there is no connection between the amount of rewrite steps n and the amount of steps needed by the TM k. Such a connection is interesting as the computational complexity of a function is defined via the amount of steps a TM takes to compute its result, i.e., computational complexity is defined as the runtime complexity of a TM. The runtime complexity of a TM is a *cost model*, indicating how expensive a computation is. Apparently such cost models are depending on the computational model. Cost models for term rewriting usually base on the amount of rewrite steps of a derivation.

A cost model is *invariant* if it has a polynomial relationship to the runtime complexity on a TM, so when it is possible to perform an equivalent computation on a TM with polynomial overhead. In [3, 2] it has been shown that the runtime complexity of a TRS is such an invariant cost model. The proof of this fact simulates term rewriting by graph rewriting, reproving results from [8, 7], and shows how graph rewriting can be implemented on a TM in polynomial time. The focus hereby lay on the control over the required resources, thus allowing an accurate characterization of the implementation. A similar result concerning derivational complexity is shown in [5]. This fact is indicated in the final picture below (where $p(n)$ denotes a polynomial in n).



After this big picture the rest of this chapter will focus on the relationship between term rewriting and graph rewriting.

Two problems induced by sharing nodes in the graph representation of a term will be discussed first. Solutions with respect to innermost rewriting will represent the results of [3]. Outermost evaluation will be inspected in a similar manner and then a solution based on the results of [2] will be given.

Definition 3.1. Let $\mathcal{R}$ be a term rewrite system. The *simulating graph rewrite system* $\mathcal{G}(\mathcal{R})$ is a graph rewrite system, containing for every rule $l \to r \in \mathcal{R}$ a graph rewrite rule $L \Rightarrow R$ such that $\mathsf{term}(L) = l$ and $\mathsf{term}(R) = r$.

The representation of a term as a termgraph is not unique. Termgraphs S and T might be different, although $\mathsf{term}(S) = \mathsf{term}(T)$.

Example 3.2. Consider the following two termgraphs S and T.

Clearly $\mathsf{term}(S) = \mathsf{term}(T)$ but $S \neq T$.

Definition 3.3. Termgraphs S and T are *isomorphic*, whenever there exists a bijective morphism $m : S \to T$. Then $\mathsf{term}(S)$ is a variant of $\mathsf{term}(T)$.

Another ambiguity is induced by different degrees of sharing.

Example 3.4. The following two termgraphs S, T again represent the same term, i.e., $\mathsf{term}(S) = \mathsf{term}(T)$ – but again $S \neq T$.

Note however that it is possible to find a morphism $m : S \to T$ but not from T to S.

Definition 3.5. Let S, T be termgraphs. T employs *more sharing* than S, denoted by $T \leqslant_m S$, if there exists a morphism $m : S \to T$ such that for all $n \in S$ holds $\mathsf{L}(n) = \mathsf{L}(m(n))$.

3.1 Problems Arising

When simulating term rewriting with graph rewriting two problems occur. For one, although a rule is applicable to a term, due to unfortunate sharing, no morphism might be found and so the term graph is in normal form. On the other hand, when rewriting a shared node in a graph, all the subterms starting from this node are rewritten parallel.

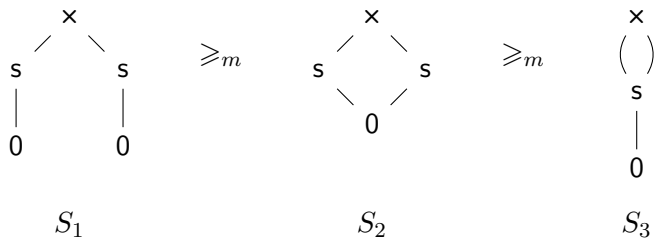
3.1.1 Problem 1: No Morphism Found

When this problem arises, a termgraph is in normal form although the corresponding term is not. To illustrate this, consider the following example.

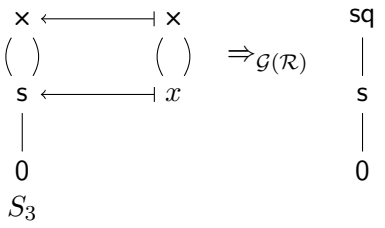
Example 3.6. Consider the TRS $\mathcal{R}$ consisting of the rule $x \times x \to \mathsf{sq}(x)$ and the term $s = \mathsf{s}(\mathsf{0}) \times \mathsf{s}(\mathsf{0})$ which admits the derivation

$$s = \mathsf{s}(\mathsf{0}) \times \mathsf{s}(\mathsf{0}) \to_{\mathcal{R}} \mathsf{sq}(\mathsf{s}(\mathsf{0})) = t \in \mathsf{NF}(\mathcal{R})$$

There are several possible graph representations of s, depicted in the following.



Clearly $\mathsf{term}(S_1) = \mathsf{term}(S_2) = \mathsf{term}(S_3) = s$ holds. But when considering the rewrite rule $x \times x \to \mathsf{sq}(x)$ it becomes obvious that is only possible to find a morphism to S_3 as seen below.



Notice that the above representation of the left hand side of the rewrite rule really is the only possible termgraph representation. Recall an invariant on termgraphs is that every variable node has to be shared. Would one dismiss this restriction, i.e., there exist two different nodes representing the same variable node the graph morphism may map those two different nodes to different subgraphs. Therefore the corresponding substitution would map the same variable to different subterms. Consequently the demand that every variable node is mapped to the same subterm induces the demand of sharing a term such that it is possible to find a morphism.

Why hasn't this problem occurred before? First through an attentive choice of examples or applying properly sharing without mentioning to do so. But secondly: This problem does not always arise. It arises whenever there is a shared variable node x on the left hand side of a rule – if a rule is not left-linear. This will be discussed further in Section 3.4.

The question remains how to properly share a termgraph so that a morphism might be found. In general it can be said that: The termgraph S has to be maximally sharing below the redex position p, i.e., $S{\restriction}\mathsf{node}(S,p)$ is maximally sharing. One might argue now, always share a termgraph as much as possible and this smoothly introduces the next problem.

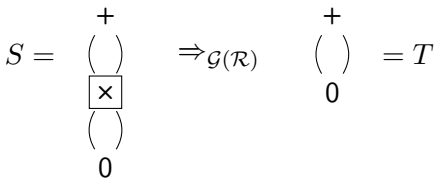
3.1.2 Problem 2: Accidental Parallel Rewriting

When one is being pedantic, he or she might want to be able to *exactly* reproduce a term rewriting step with graph rewriting.

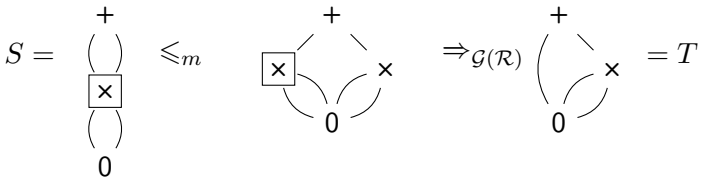
Example 3.7. So consider the TRS $\mathcal{R} = \{0 \times x \to_{\mathcal{R}} 0\}$ and the term $s = (0 \times 0) + (0 \times 0)$ with the possible rewrite step

$$s = \underline{(0 \times 0)} + (0 \times 0) \to_{\mathcal{R}} 0 + (0 \times 0) = t.$$

When the termgraph S corresponding to s is maximally sharing, there is only one rewrite step possible.



Unfortunately now $\mathsf{term}(T) \neq t$. To exactly reproduce the term rewrite step using graph rewriting some possibility to unshare a shared node has to be given. Therefore the shared node has to be reachable by a unique path and thus every shared node along the path has to be copied.



Now $\mathsf{term}(T) = t$.

As it becomes apparent, to exactly reproduce the rewrite step one needs unsharing. Now obviously the question is: Is it really necessary to exactly reproduce? The answer is yes, on the grounds that one wants an adequate model to simulate term rewriting. This becomes even more obvious when considering a non-confluent TRS.

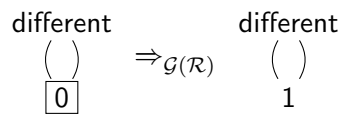
Example 3.8. Consider the TRS $\mathcal{R} =$

$$\begin{aligned} \mathsf{different}(1,0) &\to \mathsf{T} \\ 0 &\to 1 \end{aligned}$$

and the rewrite sequence

$$\mathsf{different}(\underline{0},0) \to_{\mathcal{R}} \underline{\mathsf{different}(1,0)} \to_{\mathcal{R}} \mathsf{T}.$$

The only possible graph rewrite sequence starting from a maximally sharing graph and not using unsharing is



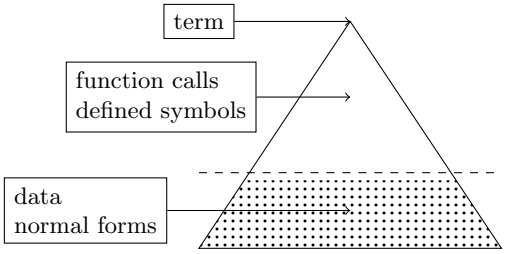
Here it is never possible to reach the normal form T.

As a second invariant of termgraph rewriting it can therefore be ascertained: The node corresponding to the redex position has to be unshared.

3.2 Adequacy of Innermost Rewriting

In this section graph rewriting with respect to the innermost evaluation strategy will be discussed first. Therefore the content from [3] will be summarized.

Innermost rewriting has by definition one for this purpose very interesting feature: A redex does not contain another redex. In this observation lies the key point of innermost graph rewriting. It allows to draw a strict line between functions and computed data – which are terms in normal form. Depicted this may look like



The redices for innermost derivations are along this line. By rewriting the data area may enlarge or shrink but the redices are always along this line.

Definition 3.9. A termgraph T is *normal form sharing* with respect to a rewrite system $\mathcal{R}$, if all nodes n with $\mathsf{term}(T\restriction n) \in \mathsf{NF}(\mathcal{R})$ are maximally shared. Moreover all nodes n with $\mathsf{term}(T\restriction n) \notin \mathsf{NF}(\mathcal{R})$ are minimally shared.

This special form of sharing allows to state the following adequacy result from [3].

Theorem 3.10. *Let $\mathcal{R}$ be a TRS and let $\mathcal{G}(\mathcal{R})$ denote the simulating graph rewrite system. Let s be a term and S be the corresponding and normal form sharing termgraph. Then $s \xrightarrow{\mathrm{i}}_{\mathcal{R}} t$ if and only if $S \overset{\mathrm{i}}{\Rightarrow}_{\mathcal{G}(\mathcal{R})} \cdot \geqslant_m T$ where T is normal form sharing*

Proof. This theorem has been shown in [3, Theorem 19]. Here only the proof idea will be given: By ascertaining that all normal forms are maximally shared, the problem of not finding a morphism is tackled by the strategy. Consider that every variable node in a rule can only be mapped to some normal form. As every normal form is shared a morphism can be found. A second condition remains to be met. The redex position has to be unshared. Assuming all the normal forms are shared new nodes will only be introduced by the right hand-side of a rule. Since rules are minimally sharing, applying a rewrite step does not introduce more sharing. On the contrary, it might require to re-establish the fact that every normal form is shared. □

Innermost rewriting corresponds to call-by-value evaluation in (functional) programs. Call-by-value requires all arguments to be *computed* values, i.e., normal forms. With this strategy however it is not possible to operate on infinite data. Suppose you want the first element of an infinite list. Call-by-value would require to first compute the whole list. This is not feasible. The opposite of call-by-value evaluation is the call-by-name strategy, which corresponds to outermost rewriting.

3.3 Adequacy of Outermost Rewriting

Call-by-name does not require the arguments of a function to be values. Thus it allows working on infinite data, like for example in Haskell. Similar to innermost rewriting, where the chosen redex does not contain another redex, one may observe that for outermost rewriting the chosen redex is not contained in another redex. Symmetrical to innermost's requirement for only sharing, one may now also expect outermost rewriting to require only unsharing to be adequate. Unfortunately this is not the case. In this section it will be shown that both operations – sharing and unsharing – are necessary to mimic outermost term rewriting.

First a counterexample to show that outermost graph rewriting with unsharing is not adequate will be given. In the next step a counterexample will illustrate that outermost graph rewriting with sharing is not adequate either. Concluding an adequate technique for outermost rewriting will be explored, implementing both sharing and unsharing based on [2].

Lemma 3.11. *There exists a TRS $\mathcal{R}$ and terms s, t such that outermost graph rewriting with unsharing is not adequate, i.e., $s \xrightarrow{o}_{\mathcal{R}} t$ such that $t \in \mathsf{NF}(\mathcal{R})$ but it is not possible to find termgraphs S, T such that $\mathsf{term}(S) = s$ and $\mathsf{term}(T) = t$ and $S \overset{o}{\Rightarrow}_{\mathcal{G}(\mathcal{R})} \cdot \leqslant_m T$.*

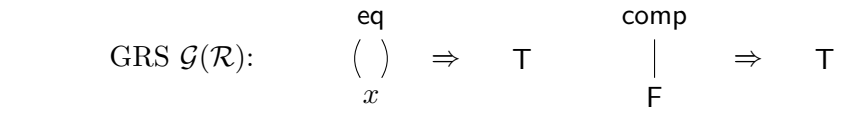
Proof. Consider the TRS $\mathcal{R}$:

$$\mathsf{eq}(x, x) \to \mathsf{T}$$
$$\mathsf{comp}(\mathsf{F}) \to \mathsf{T}$$

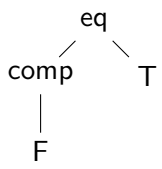
and the following outermost derivation

$$s = \mathsf{eq}(\underline{\mathsf{comp}(\mathsf{F})}, \mathsf{T}) \xrightarrow{o}_{\mathcal{R}} \underline{\mathsf{eq}(\mathsf{T}, \mathsf{T})} \xrightarrow{o}_{\mathcal{R}} \mathsf{T} = t \text{ whereas } t \in \mathsf{NF}(\mathcal{R}).$$

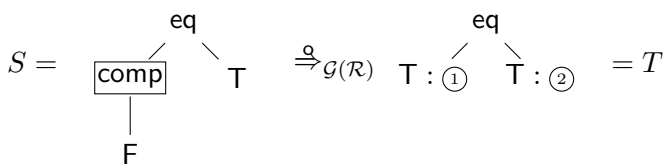
There is only one possible simulating graph representation of $\mathcal{R}$:

GRS $\mathcal{G}(\mathcal{R})$: 

The only possible termgraph S such that $\mathsf{term}(S) = s$ is



Now consider the outermost rewrite step



As no morphism can be found, T is in normal form, i.e., $T \in \mathsf{NF}(\mathcal{G}(\mathcal{R}))$, but $\mathsf{term}(T) \neq t$. □

We conclude that outermost graph rewriting without sharing is not fully adequate. In the next example it will become evident that also unsharing is a prerequisite.

Lemma 3.12. *There exists a TRS $\mathcal{R}$ and terms s, t such that outermost graph rewriting with sharing is not adequate, i.e., $s \xrightarrow{o}_{\mathcal{R}} t$ such that $t \in \mathsf{NF}(\mathcal{R})$ but it is not possible to find termgraphs S, T such that $\mathsf{term}(S) = s$ and $\mathsf{term}(T) = t$ and $S \overset{o}{\Rightarrow}_{\mathcal{G}(\mathcal{R})} \cdot \geqslant_m T$.*

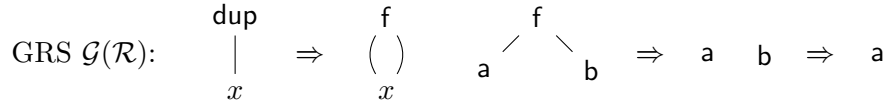
Proof. Consider TRS $\mathcal{R}$:

$$\begin{aligned} \mathsf{dup}(x) &\to \mathsf{f}(x, x) \\ \mathsf{f}(\mathsf{a}, \mathsf{b}) &\to \mathsf{a} \\ \mathsf{b} &\to \mathsf{a} \end{aligned}$$

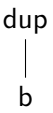
and the admitted outermost derivation

$$s = \underline{\mathsf{dup}(\mathsf{b})} \xrightarrow{o}_{\mathcal{R}} \mathsf{f}(\underline{\mathsf{b}}, \mathsf{b}) \xrightarrow{o}_{\mathcal{R}} \underline{\mathsf{f}(\mathsf{a}, \mathsf{b})} \xrightarrow{o}_{\mathcal{R}} \mathsf{a} = t$$

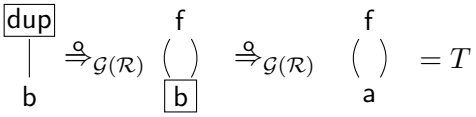
Now consider the only possible graph representation of $\mathcal{R}$

GRS $\mathcal{G}(\mathcal{R})$: 

The only possible graph representation of s is

$S =$ 

This admits the outermost derivation:

$S =$ 

$= T$

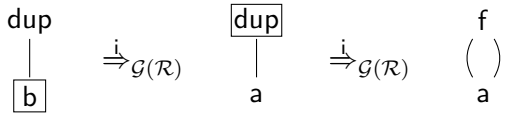
but $\mathsf{term}(T) \neq t$.
This shows that unsharing is a prerequisite for outermost graph rewriting.
□

When compared with innermost term rewriting, through the non-confluence of this system, it becomes apparent, that the strategies do not yield the same normal form. The next example is to illustrate, that innermost graph rewriting does not require unsharing.

Example 3.13. Consider the innermost rewrite sequence over the above TRS

$$s = \mathsf{dup}(\underline{\mathsf{b}}) \xrightarrow{\mathsf{i}}_{\mathcal{R}} \underline{\mathsf{dup}(\mathsf{a})} \xrightarrow{\mathsf{i}}_{\mathcal{R}} \mathsf{f}(\mathsf{a},\mathsf{a}) = t$$

Clearly, as the choice of the redex differs with innermost rewriting, unsharing does not become a prerequisite.



Next follows the inspection of outermost rewriting representing the results of [2] for full rewriting. As observed above: To rewrite a termgraph at position p the node corresponding to p has to be unshared and the subgraph starting at p has to be maximally sharing. This requires precise control over the sharing operation.

Definition 3.14. Let S, T be termgraphs and nodes $n', n \in S$. Then $S \sqsupseteq_n^{n'} T$ denotes the sharing of nodes n and n'. There exists a morphism m, with $S \geqslant_m T$, such that $m(n') = n$ and for all other nodes n_i in S holds $m(n_i) = n_i$. Define $S \sqsupset_n^{n'} T$ as $S \sqsupseteq_n^{n'} T$ and $n \neq n'$.

This allows to collapse two nodes.

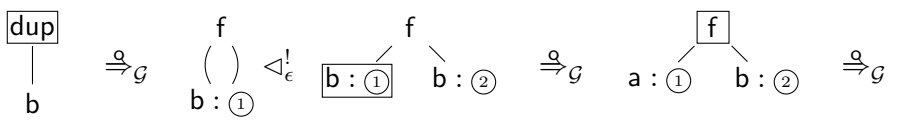
Definition 3.15. Let S, T be termgraphs and let p be a position in S. Then S shares strictly below p to T, denoted by $S \blacktriangleright_p T$ if $S \sqsupset_n^{n'} T$ for nodes n, n' strictly below p.

Definition 3.16. Let S, T be termgraphs and let p be a position. Then S unshares above p to T, denoted as $S \lhd_p T$ if $S \sqsubset_n^{n'} T$ for some unshared node n' above p.

Lemma 3.17. *Let S be a termgraph and p a position in S. If S is $\lhd_p$-minimal then the node $\mathsf{node}(S, p)$ is unshared.*

Proof. The proof can be found in [2, Lemma 4.13] □

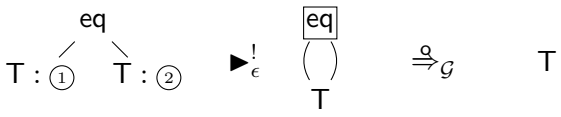
Example 3.18. Reconsider the counterexample in the proof of Lemma 3.11.



Lemma 3.19. *Let S be a termgraph and p be a position in S. If S is $\blacktriangleright_p$-minimal then $S{\restriction}\mathsf{node}(S, p)$ is maximally sharing.*

Proof. The proof can be found in [2, Lemma 4.14] □

Example 3.20. Reconsider the counter example of Lemma 3.12 – sharing nodes ① and ② is necessary to achieve the simulating graph rewrite step.



These sharing and unsharing operations now allow to state the main adequacy result from [2].

Theorem 3.21. *Let s be a term and S a termgraph such that* $\mathsf{term}(S) = s$. *Then*

$$s \to_{\mathcal{R},p} t \text{ if and only if } S \lhd_p^! \cdot \blacktriangleright_p^! \cdot \Rightarrow_{\mathcal{G}(\mathcal{R}),p} T$$

for some termgraph T with $\mathsf{term}(T) = t$.

Proof. The proof can be found in [2, Theorem 4.15] □

3.4 Adequacy of Linear Term Rewrite Systems

So far different rewrite strategies were explored and no restriction on the TRS was imposed. In the following section some restrictions on the rewrite system allow to avoid the problems induced by sharing and unsharing. Consider therefore the definition of linear.

Definition 3.22. A term t is *linear* if no variable occurs more than once in t. A termgraph T is linear if $\mathsf{term}(T)$ is linear. A rewrite rule $l \to_{\mathcal{R}} r$ is *left-linear* if l is linear. A TRS $\mathcal{R}$ is left-linear, if l is linear for all rules $l \to_{\mathcal{R}} r \in \mathcal{R}$. A TRS $\mathcal{R}$ is linear if for all $l \to_{\mathcal{R}} r \in \mathcal{R}$ holds l, r are linear.

In [1] it is shown, that left-linearity of a TRS eliminates the need for sharing. This is justified by the observation, that every variable node occurs just once in a left-linear rule. Thus always a morphism can be found, see Lemma 3.25. However, the problem of accidentally parallel rewriting remains and thus unsharing is vital for the simulation.

Theorem 3.23. *Let $\mathcal{R}$ be a left-linear TRS and $\mathcal{G}(\mathcal{R})$ be the simulating GRS. Then $s \to_{\mathcal{R},p} t$ if and only if $S \lhd_p^! \cdot \Rightarrow_{\mathcal{G}(\mathcal{R}),p} T$ where* $\mathsf{term}(S) = s$ *and* $\mathsf{term}(T) = t$.

Proof. The proof can be found in [1, Theorem 7.3.]. □

When restricting to innermost rewriting also unsharing can be dropped.

Theorem 3.24. *Let $\mathcal{R}$ be a left linear TRS and $\mathcal{G}(\mathcal{R})$ the simulating GRS, further S is a termgraph, which does not contain any shared redices. Then $s \xrightarrow{\mathsf{i}}_{\mathcal{R}} t$ if and only if $S \overset{\mathsf{i}}{\Rightarrow}_{\mathcal{G}(\mathcal{R})} T$ where* $\mathsf{term}(S) = s$ *and* $\mathsf{term}(T) = t$ *and T does not contain any shared redices.*

Proof. The proof follows the proof in [3, Theorem 19]. Note hereby that the precondition that S is in normal form sharing is omitted by the fact that TRS $\mathcal{R}$ is left linear. In T no shared redex will be introduced by the innermost rewrite step. □

This does not hold for rewriting in general and in particular not for outermost rewriting. One may be convinced by reconsidering the counter example given in the proof of Lemma 3.12. Note that the precondition of left-linearity is fulfilled.

Lemma 3.25. *Let l be a term and $s = l\sigma$ for some substitution σ. Further $\mathsf{term}(S) = s$ and $\mathsf{term}(L) = l$. If L is linear and minimally sharing then there exists a morphism $m : L \to S$.*

Proof. This is shown by induction on l. For the base case consider $l \in \mathcal{V}$, then L consists only of the root node. Set $m(\mathsf{rt}(L)) = \mathsf{rt}(S)$. For the step case consider $l = f(t_1, \ldots, t_k)$ and $s = f(t_1\sigma, \ldots, t_k\sigma)$. Further suppose $\mathsf{Succ}_L(\mathsf{rt}(L)) = [n_1, \ldots, n_k]$ and $\mathsf{Succ}_S(\mathsf{rt}(S)) = [u_1, \ldots, u_k]$. By the induction hypothesis there exist morphisms $m_1, \ldots, m_k$ with $m_i : L{\restriction}n_i \to S{\restriction}u_i$. Define the morphism $m : L \to S$ as follows. Set $m(\mathsf{rt}(L)) = \mathsf{rt}(S)$ and $m(v) = m(v_i)$ if v is in the domain of m_i. Note that the domains of $m_1, \ldots, m_k$ are pairwise disjoint as no node in L is shared, in particular no variable node due to linearity. Hence m is well defined. □

Theorem 3.26. *Let $\mathcal{R}$ be a linear TRS and $\mathcal{G}(\mathcal{R})$ be the simulating GRS. Then $s \to_{\mathcal{R},p} t$ if and only if $S \Rightarrow_{\mathcal{G}(\mathcal{R}),p} T$ where $\mathsf{term}(S) = s$ and $\mathsf{term}(T) = t$ and S, T minimally sharing.*

Proof. Following the proof of [2, Theorem 4.15] it has to be established that $\mathsf{node}(S, p) = n$ is unshared. This is easily justified, as S is minimally sharing and by definition of rewrite rule $\mathsf{L}_S(n) \notin \mathcal{V}$, thus n is not shared. Further it has to be established, that it is possible to find a morphism $m : L \to S{\restriction}n$. This holds by Lemma 3.25, as $\mathcal{R}$ is linear, in particular left linear, and S minimally sharing. Last it has to be ascertained, that T is minimally sharing. Assume T is not minimally sharing, then there is a node u with $\mathsf{L}_T(u) \in \mathcal{F}$ and u is shared. If $u \in S$ i.e., $u \in S[\Box]_p$ then S was not minimally sharing. Hence $u \in m(R')$ (where $L' \Rightarrow R'$ denotes the renaming of the rewrite step). Note that R' does not contain shared nodes as it is linear. Thus $u \notin R'$. Therefore $u \in S{\restriction}v$ for some v. Since $S{\restriction}v$ is minimally sharing the only possibility left is $u = v$. However this contradicts the right linearity of R'. □

4 Implementation

Originally designed to support the theoretical part of the thesis the program expanded steadily–now supplementing a command line mode as well as an interactive mode, which allows the user to experiment with term graph rewriting.

This original design induces a major draw-back though. The choice of the programming language was not based on its qualification for this project, but rather out of personal interest. The functional programming language Haskell has been elected. The library implementing term rewriting is currently a project of the Computational Logic Group and available online[1].

4.1 A User's View

This section first establishes the input format. Further for interested readers it explains means to execute the program.

4.1.1 Input Format

The program requires two arguments: the term rewrite system and the term to rewrite. A term rewrite system is provided by the path to a file containing the system. This term rewrite system has to conform the specification for termination problem data base (tpdb) entries also provided online[2].

Example 4.1. The TRS with the single rule $0 + x \to 0$ in conformity with the by tpdb induced specification.

```
(VAR x)
(RULES
+(0, x) -> x)
```

A term is given as a string. The distinction of variable symbols and constants is done via the argument list. Where variables do not carry any arguments, for constants the empty argument list is required.

Example 4.2. The term $x + 0$ is given as an argument as `"+(x,0())"`.

4.1.2 Command Line

The program supports a command line interface. The installation of the program is – thanks to cabal[3] – very easy: Type `$ cabal install` in the directory where you put the program.

To call the program one has to issue the following command[4]:

[1] `http://cl2-informatik.uibk.ac.at/git/?p=rewriting;a=summary`
[2] `http://www.lri.fr/~marche/tpdb/format.html`
[3] `http://www.haskell.org/cabal/`
[4] `$ graphrewrite` This will lead to the answer: `Requires at least 2 arguments, got 0.`

```
$ graphrewrite trs t
```

The arguments, the term rewrite system `trs` and the term `t`, have to be given and have to follow the specification above.

Example 4.3. A possible call to the program
`$ graphrewrite "/path/to/trs" "+(x,0())"`.

By default the program will now apply outermost graph rewrite steps to the given start term until some normal form is reached. All intermediate steps will be printed on to the console.

Furthermore some options are provided:

- `-n` will skip printing the intermediate steps.
- `-t` will apply outermost term rewriting instead of graph rewriting.

4.1.3 Interactive

The tool also supports interactive use. Therefore installation of GHCi[5], version 6.12.1 or above is required. GHCi is GHC's interactive environment. It supports evaluating expressions interactively and interprets programs. To start the program in an interactive mode, issue the command `$ ghci` in the directory where you extracted the program. A Haskell interpreter with the graph and term rewriting modules loaded will start up.

The following commands are supported. It is possible to store a rewrite system, a termgraph and a term globally so repetitive rewrite steps can be performed on them.

- `load_sys :: FilePath -> IO ()`
  Parses a term rewrite system from the given file and stores it globally as graph rewrite system and term rewrite system.
- `load_term :: String -> IO UTerm`
  Parses a term from the given string and stores it globally.
- `load_termgraph :: String -> IO TermGraph`
  Parses a termgraph from the given string and stores it globally.
- `load_term_and_termgraph :: String -> IO (TermGraph, UTerm)`
  Parses a term from the given string and stores the term and the corresponding termgraph globally.

This values once set, it is possible to reference them via the following functions.

- `term :: IO UTerm`
  Returns the globally stored term.

[5] `http://www.haskell.org/ghc/`

- `termgraph :: IO TermGraph`
  Returns the globally stored termgraph.

- `trs :: IO TermRewriteSystem`
  Returns the globally stored term rewrite system. To load a term rewrite system see the command `load_sys`.

- `grs :: IO GraphRewriteSystem`
  Returns the globally stored graph rewrite system. To load a graph rewrite system see the command `load_sys`.

This globally set values might be rewritten now. Three choices for term rewriting are available.

- `t_step :: IO UTerm`
  Performs a single rewrite step on the term set by `load_term` wrt. the trs set by `load_sys` and stores the resulting term globally.

- `t_steps :: Int -> IO UTerm`
  Performs the given amount `n` of rewrite steps on the term set by `load_term` wrt. the trs set by `load_sys` and stores the resulting term globally.

- `t_nf :: IO UTerm`
  Performs rewrite steps on the term set by `load_term` wrt. the trs set by `load_sys` until a normal form is reached and stores the resulting term globally.

Analogous it is possible to rewrite a termgraph with the corresponding functions.

- `gr_step :: IO TermGraph`
  Performs a single rewrite step on the termgraph set by `load_termgraph` wrt. the grs set by `load_sys` and stores the resulting termgraph globally.

- `gr_steps :: Int -> IO TermGraph`
  Performs the given amount `n` of rewrite steps on the termgraph set by `load_termgraph` wrt. the grs set by `load_sys` and stores the resulting termgraph globally.

- `gr_nf :: IO TermGraph`
  Performs rewrite steps on the termgraph set by `load_termgraph` wrt. grs set by `load_sys` until a normal form is reached and stores the resulting termgraph globally.

Furthermore a termgraph may employ different degrees of sharing. The program enables the user to manually share a termgraph below a position or unshare a given position.

- `sharing :: Position -> IO TermGraph` Maximally shares the termgraph set by `load_termgraph` below given position `p`.

- `unsharing :: Position -> IO TermGraph` Unshares given position `p` in the termgraph set by `load_termgraph`.

As the program is started within the GHCi-environment, all of Haskell's Prelude functions are available to the user. That is why the program allows all the above mentioned function concerning graph and term rewriting also in an advanced version. The functionality stays the same, but the arguments are given by the user. To clarify this:

- `help :: IO ()`

4.2 About the Implementation

This section briefly discusses the internals of the implementation. Here just a short introduction to the program is given, the kind reader is referred to the (haddock) documentation of the program for further insight, which can be generated by `$ cabal haddock`.

The implementation bases on the Inductive Graph library[6]. This library provides an interface for graph manipulation. When closer inspecting the graph structure it becomes apparent that the implementation of graphs is based on a map of contexts. A context contains information about ancestors, successors and the label of a node in the graph. Unfortunately a map is a poor substitute for pointers. This will become more apparent in Chapter 5. Consider the drawback when inserting and edge into the graph. The contexts of the two nodes concerned have to be fetched updated and re-inserted.

Mostly the functions of the program directly implement the theory. For two functions `foldtg`, which implements sharing, and `step`, implementing an outermost graph rewrite step, the implementation differs slightly and thus those will be discussed here.

The implementation of `foldtg` is realized by a bottom-up approach. So first all nodes with no successors are selected – the leaf nodes. Among these all nodes which are candidates for sharing are shared. Then the evaluation moves one level up, to the ancestors of the recently inspected nodes, and tries to share among those. Note that the key idea here is, that whenever a node is considered for sharing all it's successors are already maximally shared. This continues until the root node is reached. This results in a maximally sharing termgraph. Advantage to this approach is that the termgraph has to be traversed only twice – to find the leaf nodes and to share bottom up.

Next the implementation of `step` will be discussed. Recall that by Theorem 3.21 a rewrite step can be performed as

$$S \lhd_p^! \cdot \blacktriangleright_p^! \cdot \Rightarrow_{\mathcal{G}(\mathcal{R}),p} T$$

Here it is tacitly assumed that the redex position p is given. Unfortunately this is not the case, but p has to be found by repeatedly trying to find a

[6] `http://hackage.haskell.org/packages/archive/fgl/latest/doc/html/Data-Graph-Inductive-Graph.html`

morphism between a rule and some subgraph of S. As observed before, to find a morphism this subgraph has to be maximally sharing. [2, Lemma 5.8] uses the following approach to find a redex position and perform the rewrite step. To check whether a node corresponds to a redex, $S \blacktriangleright^!_p S_1$ for $p \in \mathsf{pos}(S, n)$ is computed. If n does correspond to a redex, the rewrite step is performed as $S \triangleleft^!_p \cdot \blacktriangleright^!_p S_2 \Rightarrow_{\mathcal{G}(\mathcal{R}),p} T$. Note that $\blacktriangleright^!_p$ is computed at least twice, probably more often, if several nodes have to be checked. Further observe that this approach is for full rewriting.

Since this implementation focuses on outermost rewriting the first node to be checked will always be the root node. Hence the first sharing will be $S \blacktriangleright^!_\epsilon S_1$. Obviously $S_1 \restriction p$ is maximally sharing for all p, i.e., $S_1 \blacktriangleright^!_p S_1$. Thus searching for a redex position can be done in S_1 for all nodes. Moreover the approach above discards the sharing when performing the rewrite step. Keeping the intermediate result S_1 allows to perform the whole rewrite step as

$$S \blacktriangleright^!_\epsilon S_1 \triangleleft^!_p S_2 \Rightarrow_{\mathcal{G}(\mathcal{R}),p} T$$

where p is the redex position found in S_1.

5 Evaluation of the Implementation

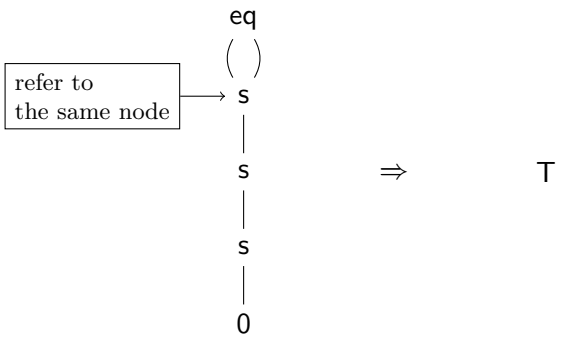
This chapter is structured in two parts – how it should be and how it turned out to be in a reality check with *this* implemented program.

5.1 Theoretical Advantages of Graph Rewriting

An advantage of graph rewriting lays on an implementational level. Consider the rule $\mathsf{eq}(x, x) \to \mathsf{T}$ and the term $\mathsf{eq}(\mathsf{s}(\mathsf{s}(\mathsf{s}(0))), \mathsf{s}(\mathsf{s}(\mathsf{s}(0))))$. To find the substitution $\sigma : x \mapsto \ldots$ it has to be ascertained, that both arguments of eq are mapped to the same by x, i.e., are identical terms. Therefore an equality check has to be performed, i.e.,

$$\begin{aligned} \mathsf{s}(\mathsf{s}(\mathsf{s}(0))) = \mathsf{s}(\mathsf{s}(\mathsf{s}(0))) &\iff \\ \mathsf{s}(\mathsf{s}(0)) = \mathsf{s}(\mathsf{s}(0)) &\iff \\ \mathsf{s}(0) = \mathsf{s}(0) &\iff \\ 0 = 0 & \end{aligned}$$

The tedious recursive equality check in term rewriting can be omitted by checking for pointer equality in a termgraph.



Not by coincidence the example was chosen to be eq. In a maximally shared termgraph $\mathsf{term}(S\restriction n) = \mathsf{term}(S\restriction m)$ implies $n = m$. Therefore by establishing that the successors of eq are referring to the same node one directly obtains that they represent the same (sub-)term.

As mentioned before: The growth a termgraph is linearly bound in the length of the rewrite sequence. This fact will be elaborated further.

Definition 5.1. Let t be a term. The size of t, $|t|$, expresses the amount of function symbols and variables in t. Let T be a termgraph. The size of T, denoted by $|T|$, refers to the amount of nodes in T, i.e., the size of the set V_T.

The following lemma from [2] illustrates one of the main benefits of graph rewriting.

Lemma 5.2. *Let S_0, T be termgraphs and $\mathcal{G}$ be a graph rewrite system and let $S_0 \lhd_p^! \cdot \blacktriangleright_p^! S$. Consider the step $S \Rightarrow_{\mathcal{G},p} T$. Then $|T| \leqslant |S| + \Delta$, where $\Delta = max\{|R| \mid L \Rightarrow R \in \mathcal{G}\}$.*

The size growth in a rewrite step of a termgraph is bound by the size of the largest right hand side of a rule in the graph rewrite system. This does not hold for term rewriting because subterms might be duplicated.

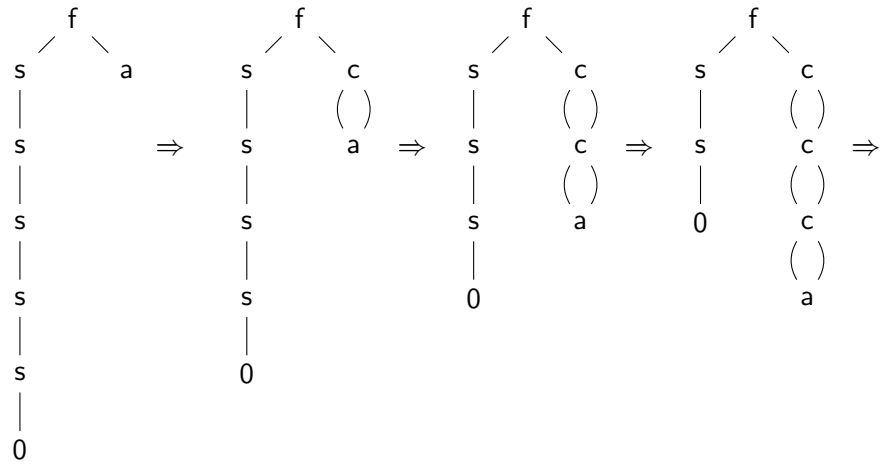
Example 5.3. To illustrate the advantage of graph rewriting consider the following TRS.

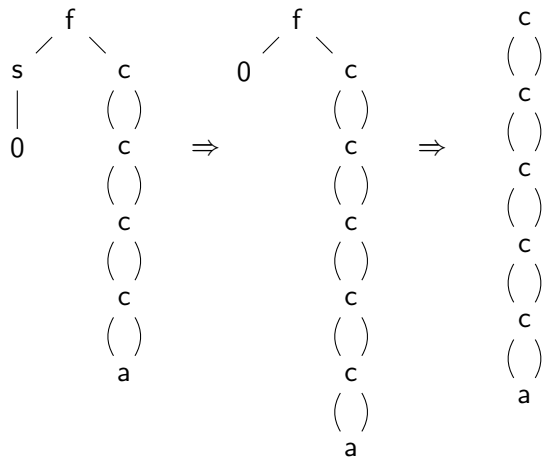
$$\begin{aligned} \mathsf{f}(\mathsf{s}(n), x) &\to \mathsf{f}(n, \mathsf{c}(x, x)) \\ \mathsf{f}(\mathsf{0}, x) &\to x \end{aligned}$$

As one can see the subterm x is duplicated (or copied, hence c) every step.

$$\begin{aligned} \mathsf{f}(\mathsf{s}^5(\mathsf{0}), \mathsf{a}) &\to \mathsf{f}(\mathsf{s}^4(\mathsf{0}), \mathsf{c}(\mathsf{a}, \mathsf{a})) \\ &\to \mathsf{f}(\mathsf{s}^3(\mathsf{0}), \mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a}))) \\ &\to \mathsf{f}(\mathsf{s}^2(\mathsf{0}), \mathsf{c}(\mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a})), \mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a})))) \\ &\to \mathsf{f}(\mathsf{s}(\mathsf{0}), \mathsf{c}(\mathsf{c}(\mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a})), \mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a}))), \\ &\qquad \mathsf{c}(\mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a})), \mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a}))))) \\ &\to \mathsf{f}(\mathsf{0}, \mathsf{c}(\mathsf{c}(\mathsf{c}(\mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a})), \mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a}))), \\ &\qquad \mathsf{c}(\mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a})), \mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a})))), \\ &\qquad \mathsf{c}(\mathsf{c}(\mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a})), \mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a}))), \\ &\qquad \mathsf{c}(\mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a})), \mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a})))))) \\ &\to \mathsf{c}(\mathsf{c}(\mathsf{c}(\mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a})), \mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a}))), \\ &\qquad \mathsf{c}(\mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a})), \mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a})))), \\ &\qquad \mathsf{c}(\mathsf{c}(\mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a})), \mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a}))), \\ &\qquad \mathsf{c}(\mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a})), \mathsf{c}(\mathsf{c}(\mathsf{a}, \mathsf{a}), \mathsf{c}(\mathsf{a}, \mathsf{a}))))) \end{aligned}$$

The same derivation using graph rewriting is pictured below.





The linear size growth of a termgraph in the length of the derivation allows a precise control over the resources occupied. Thus when implementing term rewriting via graphs the complexity of functions computed by a TRS can be analyzed. This analysis for innermost rewriting was presented in [3] and extended to full rewriting in [2]. The results are summarized below.

Definition 5.4. Let $\mathcal{F}$ be a signature and $\mathcal{R}$ be a TRS. Divide $\mathcal{F}$ into *defined* symbols $\mathcal{D} := \{f \mid \mathsf{rt}(l) = f \text{ for some } l \to r \in \mathcal{R}\}$ and $\mathcal{C} := \mathcal{F} \setminus \mathcal{D}$. Terms over $\mathcal{T}(\mathcal{C}, \mathcal{V})$ are called *constructor terms* and are collected in $\mathcal{V}al$.

Definition 5.5. A confluent and terminating TRS $\mathcal{R}$ computes an n-ary partial function $f : \mathcal{V}al^n \to \mathcal{V}al$ if there exists a defined symbol $\mathsf{f} \in \mathcal{D}$ and for all $s_1, \ldots s_k, t \in \mathcal{V}al$ holds:

$$\mathsf{f}(s_1, \ldots s_k) \to^{!}_{\mathcal{R}} t \iff f(s_1, \ldots s_k) = t$$

A polynomial relationship between the number of rewrite steps admitted by $\mathcal{R}$ and the computational complexity of the defined functions has been proven in [3] and [2]. These results will be presented in the following. The runtime complexity of a TRS is the longest derivation possibly be made in relation to the size of the start term.

Definition 5.6. The derivation length of a term s with respect to the rewrite relation $\to$ is defined as $\mathsf{dl}(s, \to) := max\{k \mid \exists t \text{ such that } s \to^k_{\mathcal{R}} t\}$.

Definition 5.7. The runtime complexity $\mathsf{rc}(n)$ of a TRS $\mathcal{R}$ is defined by $\mathsf{rc}(n) := max\{\mathsf{dl}(s, \to) \mid s = f(s_1, \ldots, s_k) \text{ and } s_1, \ldots s_k \in \mathcal{V}al \text{ and } |s| \leq n\}$. The innermost runtime complexity rc^i is defined analogous with $\xrightarrow{\mathsf{i}}_{\mathcal{R}}$.

Theorem 5.8. *Let $\mathcal{R}$ be a confluent and terminating TRS, moreover suppose $\mathsf{rc}^i = \mathsf{O}(n^k)$ for all $n \in \mathbb{N}$ and some $k \in \mathbb{N}$. The functions computed by $\mathcal{R}$ are computable in time $\mathsf{O}(n^{5(k+1)})$.*

Proof. The proof can be found in [3, Theorem 27]. □

Theorem 5.9. *Let $\mathcal{R}$ be a confluent and terminating TRS, moreover suppose $\mathsf{rc}(n) = \mathsf{O}(n^k)$ for all $n \in \mathbb{N}$ and some $k \in \mathbb{N}$ and $k \geqslant 1$. The functions computed by $\mathcal{R}$ are computable in time $\mathsf{O}(n^{7k+3})$.*

Proof. The proof can be found in [2, Theorem 6.2.]. □

5.2 Reality Check

Now the actual performance of the implementation was measured in comparison to the implementation of term rewriting mentioned in Chapter 4. The tests were performed on an ASUS EEE 1005HA with one Intel(R) Atom(TM) CPU N270 1.60GHz and 1 GB main memory. As operating system Fedora release 13 (Goddard) with the 32bit kernel in version 2.6.34 was used. The version of GHC was 6.12.1. Moreover the profiling mechanism of GHC[1] was used to get more detailed information about the behavior of the program.

An initial attempt was to compare the implementations on a simple TRS computing the square of a natural number to get an idea about the performance. The following TRS with terms of the form $\mathsf{sq}(\mathsf{s}^n(0))$ was used.

$$\begin{aligned} \mathsf{0} + y &\to y \\ \mathsf{s}(x) + y &\to \mathsf{s}(x + y) \\ \mathsf{0} \times y &\to \mathsf{0} \\ \mathsf{s}(x) \times y &\to (x \times y) + y \\ \mathsf{sq}(x) &\to x \times x \end{aligned}$$

Figure 5.1 shows the connection between the size of the input term and the run time of the program when using graph rewriting and term rewriting, respectively.

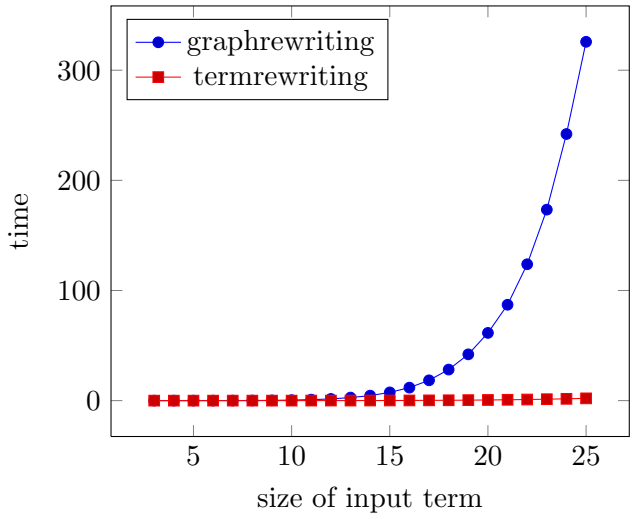


Figure 5.1: Runtime for the square function.

[1] `http://www.haskell.org/ghc/docs/latest/html/users_guide/profiling.html`

To find which operations require the resources in graph rewriting some example TRS were constructed.

First, to minimize the effort of finding a redex and sharing, trying to take full advantage of outermost rewriting consider the TRS implementing the identity function: $\mathsf{id}(x) \to x$. The expectation was that term rewriting trumps graph rewriting as this example does not exploit any benefits of graph rewriting. Nevertheless this test was made to find potential bottlenecks. In comparison with term rewriting the program runtime was like shown in Figure 5.2.

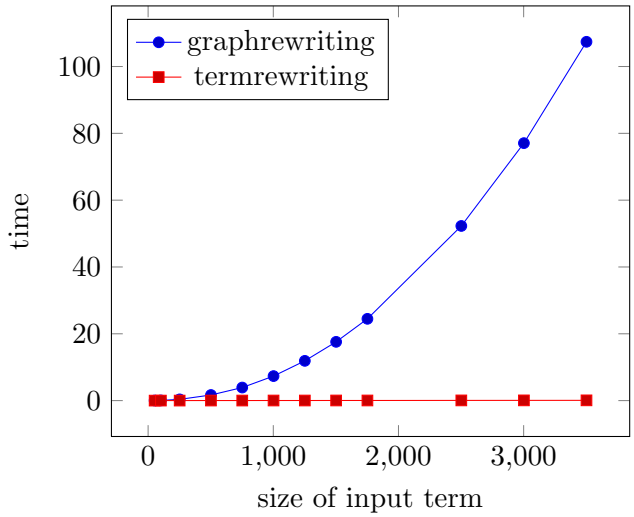


Figure 5.2: Runtime for the identity function.

When profiling with the input term $\mathsf{id}^{2500}(\mathsf{0})$ it becomes apparent that graph rewriting needs 61.2 % of the runtime for calls to the function `subgraph`, which implements garbage collection. Some 56.8 % of this 61.2 % are spent in calls to `reachable`, an implementation provided by the Graph Library to detect whether a node is reachable from a given node. A further bottleneck lies in the function `foldtg`, which implements the maximal sharing of a graph. About 37.8 % of the time are spent here. This may come as a surprise, as no actual sharing is done in the derivation. Unfortunately still a large termgraph has to be checked to ascertain this fact. The implementation of `foldtg` issues over 3 Million calls to the function `ancestors` which take up 15.9 % of the time. Further, implied by the bottom up approach of `foldtg` another 14.5 % are spent in finding leaf nodes. A possible point for improvement in future implementations is some kind of memoization to check whether a (sub-)graph is still maximally shared.

Th next example was constructed to further investigate the finding of redices and `foldtg`. The TRS considered is $x \times \mathsf{0} \to \mathsf{0}$, together with terms of the following form $\times(\times(\ldots \times(\times(\mathsf{0},\mathsf{0}),\times(\mathsf{0},\mathsf{0}))\ldots))$. Here it was expected that after the initial effort of sharing the termgraph, graph rewriting will catch up to term rewriting, which has to handle large terms. However this did not hold as the size of the term reduces in every step.

The runtime comparison is depicted in Figure 5.3.

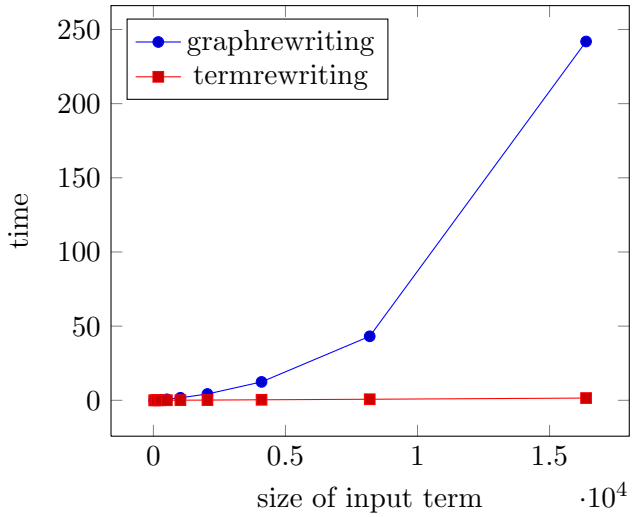


Figure 5.3: Runtime for simple multiplication.

Profiling the program shows: The call of the first rewrite step takes up 59.3 % of the programs total runtime. This is entirely due to the first call of `foldtg`, as the initial minimally sharing termgraph, which was generated from the input term, has to be shared maximally. This 59.3 % split into 39.5 % for calls to `redirect` which redirects an edge from one node to another. Hereby are 10.5 % of the time spent in `insertEdge` and 28.8 % `deleteEdge`. Another chunk in this `foldtg` call with 16.1 % is calls to `deleteNode`, which is called for over 16000 nodes. The further rewrite steps take 36.6 % of the time. Within this percentage 19 % are spent in `foldtg`, the rest of the time is evenly distributed among the various functions implementing rewriting.

Nevertheless it is possible to construct an example TRS, where graph rewriting has a slight advantage over term rewriting. Note that this TRS is very similar to the TRS considered in Example 5.3. Here the advantage, that graph rewriting does not duplicate subgraphs throughout the derivation is utilized.

$$\begin{aligned} \mathsf{f}(n) &\to \mathsf{f}'(n, \mathsf{a}) \\ \mathsf{f}'(\mathsf{s}(n), x) &\to \mathsf{f}'(n, \mathsf{dup}(x, x)) \\ \mathsf{f}'(\mathsf{0}, x) &\to x \end{aligned}$$

The runtime comparison is shown in Figure 5.4.

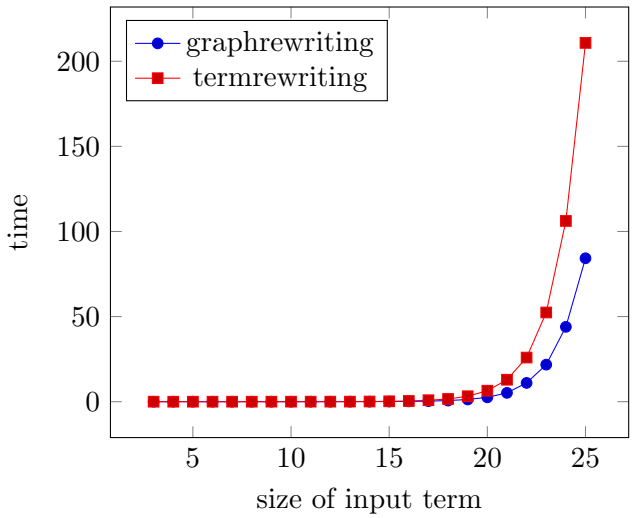


Figure 5.4: Runtime for a duplicating rule.

This statistics underline the major drawback of term rewriting which can be overcome by graph rewriting, even though the implementation is not optimized.

6 Conclusion

This thesis set out with the aim to prove adequacy of outermost graph rewriting and to find some complexity bound. As this has been established in [2] for graph rewriting in general, it was clear that outermost rewriting is adequate. The idea – or hope – was that outermost rewriting allows the redundancy of the sharing operation. In this work it has been shown that this does not hold, i.e., sharing and unsharing operations are necessary to simulate outermost term rewriting with graph rewriting. Thus the results of [2] are re-presented to give an adequate simulation.

The initial implementation of outermost graph rewriting has then evolved further. Originally designed to support the theory, to allow me deeper insight to the techniques behind graph rewriting. Therefore the choice of the functional programming language Haskell was taken out of personal interest rather than suitability. This was clearly established in the last chapter of the thesis – the evaluation of the implementation. As graphs are not a recursive data structure a functional approach was not the best one. Therefore I conclude in future I will take special care in choosing the most suitable option for matters at hand.

Bibliography

[1] Z. M. Ariola, J. W. Klop, and D. Plump. Bisimilarity in Term Graph Rewriting. *Information and Computation*, 156(1-2):2–24, 2000.

[2] M. Avanzini and G. Moser. Closing the Gap Between Runtime Complexity and Polytime Computability. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA'10*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 33–48. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2010.

[3] M. Avanzini and G. Moser. Complexity Analysis by Graph Rewriting. In *Proceedings of the 10th International Symposium on Functional and Logic Programming, FLOPS'10*, volume 6009 of *Lecture Notes in Computer Science*, pages 257–271. Springer, 2010.

[4] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, New York, NY, USA, 1998.

[5] U. D. Lago and S. Martini. Derivational Complexity is an Invariant Cost Model. In *Proceedings of the 1st International Conference on Foundational and Practical Aspects of Resource Analysis, FOPARA'09*, volume 6324 of *Lecture Notes in Computer Science*, pages 100–113. Springer, 2010.

[6] A. Middeldorp. Term Rewriting. Lecture Notes, 2009. University of Innsbruck.

[7] D. Plump. *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 1, pages 3–61. World Scientific, 1999.

[8] D. Plump. Essentials of Term Graph Rewriting. *Electronic Notes in Theoratical Computer Science*, 51:277–289, 2001.