Church-Rosser Languages and their Application to Parsing Problems

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

Dissertation

zur Erlangung des akademischen Grades Dr. rer.nat.

vorgelegt von
Dipl.-Inform. Jens Robert Woinowski
geboren in Frankfurt am Main

Tag der Einreichung: 20. August 2001
Tag der mündlichen Prüfung: 22. Oktober 2001

Referenten:
Prof. Dr. Hermann Walter, Darmstadt
Prof. Dr. Friedrich Otto, Kassel

Darmstädter Dissertationen D17

Contents

Preface

There are a lot of people who have supported me in writing this dissertation, every one of them in a special way. First of all I wish to thank Professor Hermann Walter. He has been my teacher since the time of my undergraduate studies of computer science. At that time I did not know that in the end he would be the supervisor of my dissertation. In retrospect, after working nearly five years in his group, I just can say he has been inspiring, encouraging, and supporting. And while I enjoyed academic freedom in the classical sense, he was the one who gave me the advice to reread some papers on Church-Rosser languages, leading me to the subject of my work in the last years.

I am very glad that Professor Friedrich Otto agreed to be the second supervisor of my dissertation. I thank him for interesting discussions, good questions, and helpful hints. The same holds for Gundula Niemann—I also enjoyed writing together with her.

My thanks also go to all the colleagues at the "Automata Theory and Formal Languages" group and at the computer science department of TU Darmstadt. It has been a good time working with them. Especially, Ulrike Brandt always had an open door, willing to discuss formal languages in nearly every obscure aspect possible.

Another one of those "discussion victims" has been Julia Stoll. Hopefully, she did enjoy our discussions as much as I did. Andreas Zeidler and Franziska Siebel went through the hardship of searching typographical and grammatical errors. It is astonishing how many of those errors would have survived without their help. Well, the three are very good friends, and I hope they will be in the future, too. I also want to thank all my other friends who endured my strange moods during the last six months.

Finally, I wish to express my love and gratitude to my family, especially my mother and my late father. A lot of my achievements are the result of their care and efforts.

Darmstadt, August 2001 Jens R. Woinowski

Chapter 1

Introduction

What is interesting about a language class? For the theorist, the answer could be: an elegant mode of definition, closure properties, decidability and complexity of the word problem, position in the Chomsky hierarchy [Cho59], and the likes. The more practically oriented would add: Descriptional complexity, parsing properties, including error detection and correction, the possibility to convey semantical information, and understandability. For the software engineer this all condenses into one word: tools. Yet, an open-minded engineer will admit that for the construction of useful tools very often at least some of the more theoretical properties are necessary.

As an example, consider the deterministic context-free languages: Their history was paradigmatic with respect to the above said. And their rise was fast. From Knuth's definition of LR(k)-grammars in 1965 [Knu65] to YACC by Johnson [Joh75] it only took ten years. Their advantages are well-known, e.g. deterministic parsing in linear time, easy and efficient description by grammars. And their machine model allows finding an error in an input at the leftmost postion it can be detected. On the other hand, the average user of LR(k)-parser generators probably will not *fully* understand when and why a context-free grammar is LR(k) or not, and therefore depends on the tools to detect faulty grammars and on helpful comments given by these tools.

Enter the Church-Rosser languages: Defined by McNaughton, Narendran, and Otto in 1988 [MNO88], they show a certain aesthetic elegance. In their definition, some standard techniques of formal language theory are combined. First of all, at the core, there are *confluent rewriting systems*. That is, we have a *finite alphabet*, consisting of *letters*, e.g. a set $\{a, b\}$.[1] Now a rewriting system is a set of *rules* which describe how

[1] All examples in this introductory chapter are only intended to give a rough outline of the concepts. Formal definitions will be given later.

words (strings) over that alphabet are transformed into each other. Such rules are ordered pairs of words, like (ab, a). This rule can be understood as an instruction to replace each substring ab in a word by a single letter a. Thus, the word aba would be transformed to aa, and if we took the word $abbbb$, this could be changed to aa by applying the rule four times:

$$abbbb \to abbb \to abb \to ab \to a.$$

Applying a rule is denoted by $\to$. Now consider another example, consisting of an alphabet $\{a, b, c\}$ and two rules (ab, c) and (ba, c). With those rules, the word aba can be transformed to ac and to ca. Graphically, this looks like:

$$\begin{tikzcd} & aba \arrow[dl] \arrow[dr] & \\ ac & & ca \end{tikzcd}$$

This graphical view shows both possibilities are a kind of a "dead end", after which no further rule can be used. This means, the set of the two rules given above is not *confluent.* Assume one additional rule $(c, \square)$, where $\square$ is the *empty word*, meaning that c is deleted. Then we would obtain:

$$\begin{tikzcd} & aba \arrow[dl] \arrow[dr] & \\ ac \arrow[dr] & & ca \arrow[dl] \\ & a & \end{tikzcd}$$

Informally, confluence means: Whenever two or more different rules can be applied to a word, it is always possible to use some more rules, such that in the end the result is the same.

But what does *in the end* mean? Consider the two rules (ab, ba) and (ba, ab). Starting with the word ab, one could infinitely often apply the two rules after one another:

$$ab \to ba \to ab \to ba \to \dots$$

This leads to the second essential of the Church-Rosser language definition: The rewriting systems used are required to *reduce the length* of the words they transform by at least one letter. Our first two examples have this property. Thus, as one easily sees, the process of replacing substrings will always terminate. As Book showed, confluence is checked in this case with quite an easy algorithm [Boo82], a newer and more efficient algorithm can be found in [KKMN85].

A third concept is to use *variables* which store information, but may not belong to a valid input. The set of these variables is called *nonterminals*, in contrast to the possible input letters, which are called *terminals*. Suppose we have the terminals $\{[,], a\}$ and a singleton set of nonterminals $\{D\}$. Now take the rules $([a], D)$, $([D], D)$, and (DD, D). Then we can transform all bracket expressions where all inner bracket pairs contain one a into the single letter D, for example:

$$[[a][a]][a] \to [D[a]][a] \to [D[a]]D \to [DD]D \to [D]D \to DD \to D.$$

The possibility to distinguish terminals from nonterminals is a central concept, both in formal language theory in general and for the definition of Church-Rosser languages. For example, in the definition of a Church-Rosser language L there is always a distinguished single letter (usually Y), and an input is accepted (i.e. it belongs to the language L), if it is possible to transform it to that special letter.

Finally, *detecting word ends* is an important problem. The last feature used in the definition of Church-Rosser languages is to mark both ends with words of nonterminals before the rewriting rules are applied. Continuing our last example, we add the nonterminal \$ and the rule $(\$D\$, Y)$. Then every such correctly bracketed input could be transformed to a single Y after adding a \$ at both ends, but no other words. We give two final examples:

1. $\$[[a]]\$ \to \$[D]\$ \to \$D\$ \to Y$, so $[[a]]$ is a correct input and
2. $\$][a]\$ \to \$]D\$$; no further rules can be applied: Therefore $][a]$ is an input which does not belong to the defined language.

These examples show that the concept of Church-Rosser languages is very intuitive. Furthermore, the definition of these languages and the

mechanism of deciding whether a word belongs to a Church-Rosser language or not are closely related to each other: In contrast to the Chomsky hierarchy, the difference between a corresponding automaton model and the definition of the language class is marginal. A very important property of this language class is that its word problem can be decided in deterministic linear time [MNO88]. In addition, they fully contain the deterministic context-free languages, but also other (context-sensitive) languages, and therefore, they are a generalization of the deterministic context-free languages.

Another interesting language class in the context of our investigations are the *growing context-sensitive languages*, defined in 1986 by Dahlhaus and Warmuth [DW86]. These languages are defined by grammars whose productions are strictly length-increasing (also called strictly monotone grammars). Buntrock and Loryś showed that they have very interesting closure properties [BL92], [BL94]. In his Habilitationsschrift Buntrock did a detailed examination of the growing context-sensitive languages. For example, he showed that this language class also is a complexity class [Bun96].

Niemann and Otto proved that the Church-Rosser languages and the growing context-sensitive languages are closely related to each other: The former are the deterministic variant of the latter. Both language classes can be characterized by so-called *shrinking two pushdown store automata,* sTPDA, which are a generalization of automata with one pushdown store and a more restricted model than Turing machines. Where the general (nondeterministic) sTPDA's exactly accept growing context-sensitive languages, the deterministic ones are the machine model of the Church-Rosser languages.

In their proof, Niemann and Otto also found and used another result which is very important: If the rules of the defining rewriting system are allowed to be *weight-reducing*, the descriptive power is not enhanced. That means, to each letter of the alphabets, a weight (a natural number) is assigned by a *weight-function*, and the weight of a word is the sum of the weight of its letters. Then, a rewriting system is weight-reducing if there exists a weight-function such that the weight of the left-hand side of each rewriting rule is greater than that of the corresponding right-hand side. The possibility to use weight-reduction instead of length-reduction is very handy for proofs and constructions.

In the next chapter we give a detailed and formal overview of the relevant definitions and results.

The main focus of our investigation is to show that *under some restrictions* one of the most important problems for "real life" use of a language class can be solved for the class CRL: Given a definition of a Church-Rosser language L and a word w, is it possible to decide whether w is a prefix of a word in L? Deciding this question is used in parsers (and compilers) to determine at which position in an erroneous input an error occurs. In general, this question is undecidable, because every recursively enumerable language is the homomorphic image of a Church-Rosser language. This so-called *basis property* was proved by Otto, Katsura, and Kobayashi [OKK97].[2] So at least some restrictions will be necessary.

One of these restrictions we use is merely syntactical. The rules of the rewriting system in the definition of a Church-Rosser language are required to be length- or weight-reducing without further restrictions. This means that rules like (abb, ca) are allowed. In contrast to this, we require the rules to look almost like context-sensitive rules. That is, we want them to be either deleting rules like $(c, \square)$ in the example above, or to be *swapped* context-sensitive rules, being of the form

$$(uvw, uxw)$$

where u and w are words of arbitrary length, v is a word with at least one letter, and x is a single letter. We call a rewriting system with such rules *context-splittable.*[3]

As we will show, for every Church-Rosser language it is possible to give a definition using a weight-reducing rewriting system which is context-splittable. In order to do so we introduce a technique we call *weight-spreading,* which is inspired by Niemann's and Otto's work [NO98].

This result is not only introducing a technically useful normal-form. It also implies that the Church-Rosser languages can be described by *weight-increasing context-sensitive grammars.* Such a grammar is not only weight-increasing, but also acyclic. That means, if one removes

[2] Salomaa speaks of homomorphic characterizations of the recursively enumerable languages [Sal73].

[3] We impose some further syntactical restrictions which we do not discuss in this introduction.

all contexts from the rules of such a grammar, a context-free grammar remains which is cycle-free.[4] This is an important characterization of Church-Rosser languages. The details are discussed in the third chapter.

Using this result, we give a construction which allows to describe at least some prefix languages of Church-Rosser languages within the same language class. Basically, this construction cuts of (parts of) the right-hand side context and replaces the middle part of the original rewriting system rules, thus generating new rules. Unfortunately (and unavoidably), this construction does not always work correctly. Deciding if it works correctly for a specific Church-Rosser language definition could be a difficult or unsolvable problem. Therefore, we introduce an expansion of the systems defining Church-Rosser languages. For these systems it is possible to give at least decidable and sufficient conditions for correctness. One further benefit of this expansion is that whenever a prefix is correctly accepted it is possible also to identify automatically one or more correct completions to a full word. The prefix construction is described in the fourth chapter.

In the fifth chapter, the bridge to the more practical realm is built. The first part of this chapter contains an examination of deterministic context-free languages via shift/reduce-parsers for LR(k)-grammars and their connection to Church-Rosser languages. Especially, some consequences of the prefix construction on Church-Rosser language systems which are derived from shift/reduce-parsers are discussed.

The second part of the fifth chapter describes an experimental development environment for Church-Rosser languages. This environment is based on the first prototypes of the algorithms and constructions described in this dissertation, which where developed in LISP. Following these prototypes, Thorsten Rottschäfer programmed a more user friendly tool in his diploma thesis [Rot00]. With this tool it is possible to get some more practical experiences with Church-Rosser language systems.

We end with a short conclusion.

[4] This result also can be expanded to general growing context-sensitive language [NW01a].

Chapter 2

Basic Definitions and Theorems

In this chapter basic definitions for (confluent) string rewriting are given. These definitions mostly follow the monographs of Jantzen [Jan88] and of Book and Otto [BO93]. Both monographs summarize some of the fundamental definitions and results for general *reduction systems*. Since we will not consider the latter, all of the definitions and results used here are stated only for the special case of *string rewriting systems*. Furthermore, we have a short look at *growing context-sensitive languages*, which were defined by Dahlhaus and Warmuth [DW86] and examined by Buntrock [Bun96], and the related automata, *(shrinking) two pushdown store automata* ([Boo82], [BO93], and [Bun96]). This chapter concludes with the languages which are our main focus of investigation, the *Church-Rosser languages*, defined by McNaughton, Narendran, and Otto [MNO88].

2.1 String rewriting

In this section the usual notations for string rewriting are introduced.

Definition 2.1. *An* alphabet Σ *is a finite nonempty set. A* word w *is an element of the free monoid over an alphabet* Σ*:* $w \in \Sigma^*$*. If we want to address single letters of* w*, we write* $w = a_1 a_2 \cdots a_n$ *with* $a_i \in \Sigma$ *for* $1 \leq i \leq n$*. Then* n *is the* length *of* w*, also denoted with* $|w|$*. The* empty word *has length zero and is written as* $\Box$*. A* language L *is a subset of the free monoid:* $L \subseteq \Sigma^*$*.*

Definition 2.2. *Let* $w = a_1 \cdots a_n$*. Then the set of* (proper) prefixes *is defined as Pref*$(w) := \{a_1 \cdots a_i \mid i \leq n\}$ $(\mathrm{Pref}(w) \setminus \{w\})$*. Accordingly, the set of* (proper) suffixes *is Suff*$(w) := \{a_i \cdots a_n \mid 1 \leq i\}$ $(\mathrm{Suff}(w) \setminus \{w\})$*.*

Definition 2.3. *A* string rewriting system (also called semi-Thue system or rewriting system) *R on Σ is a subset $R \subseteq \Sigma^* \times \Sigma^*$. An element $(u, v) \in R$ is called a* (rewriting) rule. *R may be finite or infinite. Usually, we will only use finite rewriting systems. Therefore, when not explicitly stated otherwise, we will only speak of finite rewriting systems.*

Definition 2.4. *Let R be a rewriting system on Σ. Then the* rewriting relation *between words in Σ^* is defined as*

$$\underset{R}{\longrightarrow} := \{(xuy, xvy) \mid x, y \in \Sigma^*, (u, v) \in R\}.$$

This is also called a (single) rewriting step. *The reflexive and transitive closure is $\underset{R}{\longrightarrow}^*$.*
If $u \in \Sigma^*$ and $u \underset{R}{\longrightarrow}^* v$, v is called descendant of u. *For $u \in \Sigma^*$ every v with $u \underset{R}{\longrightarrow} v$ is called a* direct descendant of u.
If we want to speak about a rewriting step with a specific rule $r \in R$, we use $\underset{r}{\longrightarrow}$. For sequences of rules $s \in R^*$ we use $\underset{s}{\longrightarrow}^*$.
If R is known it will be omitted, and we write $\rightarrow$ or $\rightarrow^*$. In order to show more details about the rule used for a rewriting step we sometimes use $xuy \underset{(u,v)}{\longrightarrow} xvy$ with $x, y \in \Sigma^*$ and $(u, v) \in R$.

Definition 2.5. *If R is a rewriting system, then R^{-1} is the* inverse *rewriting system of R obtained by*

$$R^{-1} := \{(v, u) \mid (u, v) \in R\}.$$

Then $T := R \cup R^{-1}$ is a Thue system induced *by R. Since the rewriting relation $\underset{T}{\longrightarrow}$ of T is symmetric we use*

$$\underset{T}{\longrightarrow} = \underset{T}{\longleftrightarrow} = \underset{R}{\longleftrightarrow} \quad \text{and} \quad \underset{T}{\longrightarrow}^* = \underset{T}{\longleftrightarrow}^* = \underset{R}{\longleftrightarrow}^*,$$

henceforth obtaining the reflexive, transitive, and symmetric closure of $\underset{R}{\longrightarrow}$.

Definition 2.6. *For a rewriting system R on Σ and a word $w \in \Sigma^*$ we denote the* congruence class *of w with respect to R as*

$$[w]_R := \{w' \mid w' \in \Sigma^* \wedge w \underset{R}{\longleftrightarrow}^* w'\}.$$

The index R will be omitted if R is obvious from the context.

Definition 2.7. *Let R be a rewriting system on Σ. Then*

- *$dom(R) := \{u \mid u \in \Sigma^* \wedge \exists v \in \Sigma^* : (u, v) \in R\}$ (domain of R).*
- *$range(R) := \{v \mid v \in \Sigma^* \wedge \exists u \in \Sigma^* : (u, v) \in R\}$ (range of R).*
- *For each $u \in dom(R)$ let $range(u) := \{w \mid w \in \Sigma^* \wedge (u, w) \in R\}$ (range of u).*
- *Also, for each $v \in range(R)$ let $dom(v) := \{u \mid u \in \Sigma^* \wedge (u, v) \in R\}$ (domain of v).*

Definition 2.8. *Let R be a rewriting system on Σ. Then the* word problem for *R is defined as the following question.*

Instance: *$w, w' \in \Sigma^*$.*

Question: *are w and w' in the same congruence class with respect to R, i.e. $[w]_R = [w']_R$ or, equivalently, $w \underset{R}{\longleftrightarrow}^* w'$?*

Theorem 2.1. *There exists a finite rewriting system R such that the word problem for R is undecidable [BO93, theorem 2.5.9.].*

Definition 2.9. *Let R be a rewriting system on Σ and $w \in \Sigma^*$. If there is no $w' \in \Sigma^*$ such that $w \underset{R}{\longrightarrow} w'$ then w is* irreducible. *The set of all irreducible words with respect to R is denoted with $\textsc{Irr}(R)$.*

Definition 2.10. *Let R be a rewriting system on Σ and $w, w' \in \Sigma^*$. If $w \underset{R}{\longleftrightarrow}^* w'$ and $w' \in \textsc{Irr}(R)$ then w' is a* normal form *of w.*

If for all such w and w' with $w \underset{R}{\longleftrightarrow}^* w'$ and $w' \in \textsc{Irr}(R)$ there does not exist a $w'' \neq w'$ with $w \underset{R}{\longleftrightarrow}^* w''$ and $w'' \in \textsc{Irr}(R)$ then all words which have an irreducible descendant in Σ^* have unique *normal forms. Then we can identify the congruence class of w with w' and therefore write $w' = [w]_R$. Again, the index R can be omitted if R is obvious from the context.*

Remark 2.1. Since in the last definition no further restriction on R are made, it is possible that there exist words without a normal form because they have no irreducible descendant. On the other hand, there may be words with more than one normal form.

2.2 Automata with two pushdown stores

In this section we describe a machine model which can be used in various ways in the string rewriting context: Automata with two pushdown stores (i.e., two stacks). By restrictions on the model different language classes can be described.

There is a long tradition in the use of automata that have two pushdown stores. Since there is an overview in [Bun96] we will not go into the details of this history. Considering automata which compute problems of string rewriting, [Boo82] and [BO93] are important references.

The following definition is according to [Bun96].

Definition 2.11. *A* two pushdown store automaton (TPDA) *is a non-deterministic automaton with two pushdown stores which is defined by a 7-tuple:* $M = (Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$.

Q *is a finite* state set.
Σ *is the* input alphabet*, also called* terminal alphabet.
Γ *is the* work alphabet. *We always assume* $\Sigma \subseteq \Gamma$.
$\Gamma \setminus \Sigma$ *is the alphabet of* nonterminals.
$q_0 \in Q$ *is the* initial state.
$\perp \in \Gamma \setminus \Sigma$ *is a special nonterminal, called* bottom symbol.
$F \subseteq Q$ *is the set of* final states.
δ *is a total mapping which maps* $(Q \times \Gamma \times \Gamma)$ *into finite subsets of* $(Q \times \Gamma^* \times \Gamma^*)$. *It is called the* program of M.

Before we discuss the semantics of this definition, we want to point out the following:

Remark 2.2. For the notation of the program of an automaton there is an alternative: One can use a rewriting system instead of a function

[Sal73]. We do not use the latter because it somewhat blurs the difference between automata as recognition devices and rewriting systems as defining devices. In the context of grammars this is a very important similarity, whereas for Church-Rosser languages the difference between definition and recognition already is very small, as will be seen in section 2.5. Therefore, instead of using the similarity we want to emphasize the differences by using functions as programs.

Remark 2.3. It may look slightly confusing that there is only one bottom-symbol (which will be used both for the left-hand and right-hand store). We follow [Bun96] in doing so.

Remark 2.4. It is not necessary to restrict the domain of δ, one also could allow it to be $(Q \times \Sigma^* \times \Sigma^*)$, but this would complicate the definition of a deterministic TPDA.

Remark 2.5. Sometimes it is convenient only to define δ partially.

Assuming that Σ and Q are disjoint, we can write *configurations* of M as words uqv from $\Gamma^* Q \Gamma^*$, $u, v \in \Gamma^*$, and $q \in Q$. We call the stores *left-hand* and *right-hand* stores. The bottom of the left-hand store is the leftmost letter of the configuration, its top is the last letter before the q, which is the actual state of the automaton. The right-hand store is accordingly: its top is the first letter after the q, its bottom the last letter of the configuration. Therefore, the left-hand store is written down backwards. For a configuration $uxqyu$ such that $q \in Q$, $x, y \in \Gamma$, and $u, v \in \Gamma^*$, x and y are called *top part* of the left- and right-hand pushdown store, respectively.

Let w be an input. In the beginning of a computation, in the left-hand store there is only the bottom symbol $\perp$. In the right-hand store is w and the bottom symbol. So, *initial configurations* are of the form $\perp\ q_0 w\ \perp$. Now we define single *computation steps*. Assume $(q, A, B) \in Q \times \Gamma \times \Gamma$ and:

$$\delta(q, A, B) = \{(q_1, u_1, v_1), (q_2, u_2, v_2), \ldots, (q_m, u_m, v_m)\}$$

with

$$q, q_1, \ldots, q_m \in Q; A, B \in \Gamma; u_1, \ldots, u_m, v_1, \ldots v_m \in \Gamma^*.$$

Assume further the automaton to be in state q with A and B being on top of the left-hand and right-hand stores, respectively. Then the automaton can, for arbitrary $i, 1 \leq i \leq m$, change into state q_i and rewrite A with u_i and B with v_i. Therefore we define the relation $\underset{M}{\vdash}$ which describes single computation steps as:

$$u'AqBv' \underset{M}{\vdash} u'u_iq_iv_iv'(u', v' \in \Gamma^*).$$

With $\underset{M}{\vdash}^*$ we denote the reflexive, transitive closure of $\underset{M}{\vdash}$. If we want to give an explicit number of computation steps, we use $\underset{M}{\vdash}^m$. The M in the relation symbols can be omitted if the automaton is obvious.

Now the languages accepted with empty stores or final state are defined as:

$$\begin{aligned} N(M) &:= \{w \in \Sigma^* \mid \exists q \in Q : \quad \perp q_0 w \perp \underset{M}{\vdash}^* q \quad \} \text{ (empty stores)} \\ L(M) &:= \{w \in \Sigma^* \mid \exists q \in F, w' \in \Gamma^* : \perp q_0 w \perp \underset{M}{\vdash}^* w'q \} \text{ (final state).} \end{aligned}$$

Note that the condition for acceptance with final state, which forces the right-hand store to be empty, implies that the complete word has been read. Furthermore we can assume that the bottom symbol $\perp$ always appears only at the bottom ends of the stacks. Obviously, this is no limiting restriction for the TPDA.

Furthermore, we define the minimal number of computation steps necessary to accept a single word:

Definition 2.12. *Let $w \in N(M)(w \in L(M))$ then w* is accepted with empty stores (with final state) in time n *if*

$$\begin{aligned} n &= \min\{m \in \mathbb{N} \mid \perp q_0 w \perp \underset{M}{\longrightarrow}^m q, q \in Q\} \\ (n &= \min\{m \in \mathbb{N} \mid \perp q_0 w \perp \underset{M}{\longrightarrow}^m w'q, w' \in \Gamma^*, q \in F\}). \end{aligned}$$

For comparing functions, we use the usual notation:

Definition 2.13. *Let $t : \mathbb{N} \to \mathbb{N}$ a function. Then $O(t(n))$ is defined in the following way:*

$$\begin{aligned} O(t(n)) := \{\, t' : \mathbb{N} \to \mathbb{N} \mid & \\ & \exists c_1, c_2 \in \mathbb{N}, c_1, c_2 > 0 : \\ & \forall n \in \mathbb{N} : t'(n) \leq c_1 \cdot t(n) + c_2 \}.^1 \end{aligned}$$

Definition 2.14. *Let M be a TPDA and $t : \mathbb{N} \to \mathbb{N}$. We say M* accepts with empty stores (with final state) in time $O(t(n))$ *if there exists a function $t' : \mathbb{N} \to \mathbb{N}$, $t'(n) \in O(t(n))$ such that for all $w \in N(M)$ (or $w \in L(M)$), and corresponding $n = |w|$, there exists an m such that w is accepted in time m and $m \leq t'(n)$.*

Definition 2.15. *A TPDA M with program δ is called* deterministic *if δ maps into sets with only one element, the abbreviation is DTPDA.*

Since without any restriction this machine model is as powerful as a Turing machine, in most cases one needs a suitable restriction. Therefore, weight functions are used:

Definition 2.16. *Let Γ be a finite set. A weight function on Γ is a function $\varphi : \Gamma \to \mathbb{N} \setminus \{0\}$.*

We extend φ to a homomorphism $\varphi : \Gamma^* \to \mathbb{N}$ by defining: $\varphi(\square) := 0$ and $\varphi(w \cdot x) := \varphi(w) + \varphi(x)$ for $w \in \Sigma^*$ and $x \in \Sigma$.

Definition 2.17. *A TPDA $M = (Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$ is called* shrinking (sTPDA) *if there exists a weight function φ on $(\Gamma \cup Q)^*$ such that for all $q, q' \in Q; A, B \in \Gamma$ and all $u, v \in \Gamma^*$:*

$$\delta(q, A, B) \ni (q', u, v) \Longrightarrow \varphi(AqB) > \varphi(uq'v).$$

In this case we call φ a weight function for M. *If M is deterministic, we abbreviate this with sDTPDA.*

Definition 2.18. *A TPDA $M = (Q, \Sigma, \Gamma, \delta, q_0, \perp, F)$ is called* bounded (bTPDA) *if there exists a weight function φ on $\Gamma \cup Q$ such that for all $q, q' \in Q; A, B \in \Gamma$ and all $u, v \in \Gamma^*$:*

[1] In fact, $O(t(n))$ is an abbreviation of the more proper notation $O(\lambda n.t(n))$.

$$\delta(q, A, B) \ni (q', u, v) \Longrightarrow \varphi(AqB) \geq \varphi(uq'v).$$

We call φ a weight function for M. *If M is deterministic, we abbreviate this with bDTPDA.*

For the sake of completeness we cite the following result from [Bun96] (Satz V.3):

Theorem 2.2. *Acceptance with empty stores or final state is equally powerful. Let n be the length of input words and $t : \mathbb{N} \to \mathbb{N}$.*

(i) A language is accepted by a TPDA with empty stores in time $O(t(n))$ if and only if it is accepted by a TPDA with final state in time $O(t(n))$.
(ii) A language is accepted by a bounded TPDA with empty stores in time $O(t(n))$ if and only if it is accepted by a bounded TPDA with final state in time $O(t(n))$.
(iii) A language is accepted by a shrinking TPDA with empty stores if and only if it is accepted by a shrinking TPDA with final state.

All three results hold both in the deterministic and in the nondeterministic case.

The proofs in [Bun96] for the three parts are based on appropriate simulations of M with another TPDA M'.

2.3 Confluence and normal forms

We now summarize some basics of confluent string rewriting.

Definition 2.19. *Let R be a rewriting system on Σ.*

(a) R is confluent *if and only if for all $w, x, y \in \Sigma^*$, $w \to^* x$ and $w \to^* y$ imply that there exists $z \in \Sigma^*$ with $x \to^* z$ and $y \to^* z$.*
(b) R is locally confluent *if and only if for all $w, x, y \in \Sigma^*$, $w \to x$ and $w \to y$ imply that there exists $z \in \Sigma^*$ with $x \to^* z$ and $y \to^* z$.*

(c) *R has the* Church-Rosser property *if and only if for all $x, y \in \Sigma^*, x \leftrightarrow^* y$ there exists $z \in \Sigma^*$ with $x \rightarrow^* z$ and $y \rightarrow^* z$.*

We also say a rewriting system *"is Church-Rosser"* if it has the Church-Rosser property.

Lemma 2.1. *Let R be a rewriting system. Then R is Church-Rosser if and only if it is confluent.*

A proof for general reduction systems can be found in [BO93, lemma 1.1.7].

Corollary 2.1. *Let R be a rewriting system on Σ. Then for each $x \in \Sigma^*$, $[x]$ has at most one normal form if R is confluent [BO93, corollary 1.1.8].*

Definition 2.20. *Let R be a rewriting system on Σ. The relation $\rightarrow$ is* Noetherian *if there is no infinite sequence of words $x_0, x_1, \ldots \in \Sigma^*$ such that for all $i \geq 0 : x_i \rightarrow x_{i+1}$. We also speak of a* terminating *system R in this case.*

Lemma 2.2. *If R is a terminating rewriting system on Σ then for all $x \in \Sigma^*$ $[x]$ has at least one normal form.*

See [BO93, lemma 1.1.10] for a proof.

Even if a rewriting systems is confluent and terminating there is a certain degree of ambiguity, because a word can have more than one direct descendant. Furthermore, there can be different reductions to a normal form. Leftmost (or rightmost) reductions are one possibility to deal with this ambiguity. Book and Otto give a definition for leftmost reductions and prove that they exist for finite terminating (not necessarily confluent) rewriting systems [BO93]:

Definition 2.21. *Let R be a rewriting system on Σ. A reduction $w \rightarrow z$ is* leftmost, *denoted $w \xrightarrow{\text{lm}} z$ if the following condition holds: if $w = x_1u_1y_1$, $z = x_1v_1y_1$, and $(u_1, v_1) \in R$, and also $w = x_2u_2y_2$ and $(u_2, v_2) \in R$, then*

- *x_1u_1 is a proper prefix of x_2u_2,*
  or
- *$x_1u_1 = x_2u_2$ and x_1 is a proper prefix of x_2,*
  or
- *$x_1 = x_2$ and $u_1 = u_2$.*

Let $\xrightarrow{\text{lm}}{}^*$ denote the reflexive transitive closure of $\xrightarrow{\text{lm}}$.
The relations $\xrightarrow{\text{rm}}$, and $\xrightarrow{\text{rm}}{}^*$ are defined in the symmetric way.

Definition 2.22. *Let R be a rewriting system on Σ. Let $s \in R^*$ be a sequence of rules, $s = r_1 \cdot r_2 \cdots r_n$ for $n \in \mathbb{N}$. If for $w, w_1, w_2, \ldots, w_n \in \Sigma^*$*

$$w \xrightarrow[r_1]{\text{lm}} w_1 \xrightarrow[r_2]{\text{lm}} w_2 \xrightarrow[r_3]{\text{lm}} \cdots \xrightarrow[r_{n-1}]{\text{lm}} w_{n-1} \xrightarrow[r_n]{\text{lm}} w_n$$

holds, we call this a leftmost reduction of w with s to w_n. *Rightmost reductions are defined accordingly.*

Remark 2.6. There is a fundamental difference between the definition of leftmost (rightmost) reductions as in definition 2.22 and $\xrightarrow{\text{lm}}{}^*$ ($\xrightarrow{\text{rm}}{}^*$). The former is conserving all information about the single reductions steps necessary in the sequence s, whereas the latter simply states the existence of such a reduction.

Remark 2.7. It is important that this definition of leftmost reductions does not always preserve congruence classes for rewriting systems in general:

$$\exists R \in \Sigma^* \times \Sigma^*, w \in \Sigma^* : \{w' \mid w \xrightarrow[R]{\text{lm}}{}^* w'\} \subset [w]_R,$$

where $\subset$ denotes the *true subset relation.* Hotz gives a definition for leftmost reductions which does not bear this restriction [Hot66]. Since for confluent and terminating rewriting systems this problem does not occur, we refer the interested reader to Hotz's article.

With the following theorem we show that for each w an irreducible normal form w' can be found by leftmost reductions.

Theorem 2.3. *Let R be a finite rewriting system on the alphabet Σ. Assume that R is terminating (i.e. $\rightarrow$ is Noetherian). There is an algorithm to solve the following problem:*

Instance: *a string $w \in \Sigma^*$.*

Result: *an irreducible string w' such that $w \xrightarrow{\text{lm}}{}^* w'$.*

Proof. [BO93] For the proof, we are going to use a DTPDA. In order to make the program deterministic, the first step is some preprocessing of R to a new system R'. This preprocessing does not depend on the input w.

Construct R' as follows: For each $u \in \text{dom}(R)$ choose the rule with the lexicographically[2] smallest v such that $(u, v) \in R$ and add (u, v) to R'. Therefore, $\text{dom}(R') = \text{dom}(R)$ and $\textsc{Irr}(R') = \textsc{Irr}(R)$. Furthermore $\xrightarrow[R']{\text{lm}}{}^* \subseteq \xrightarrow[R]{\text{lm}}{}^*$.

Since R is finite, R' is finite, so there exists a $t = \max\{|u|, |v| \mid (u, v) \in R'\}$. Because R is terminating by hypothesis and $R' \subseteq R$, R' is terminating, too.

The DTPDA M we are constructing will use a loop that contains three parts. The bottom symbol $\perp$ is a new symbol not appearing in Σ. Assuming that at the entry point of the loop the configuration is $\perp x_1 q x_2 \perp$, for $q \in Q; x_1, x_2 \in \Sigma^*$, then at the end point of the loop (after the last operation within the loop, in contrast to the loop exit point, which is the end of the computation) the configuration is $\perp y_1 q' y_2$, for $q' \in Q; y_1, y_2 \in \Sigma^*$ and $x_1 x_2 \xrightarrow[R']{\text{lm}}{}^* y_1 y_2$. Because of this invariant, also $w \xrightarrow[R]{\text{lm}}{}^* y_1 y_2$ holds.

The three parts of the loop are:

(i) **READ** (*loop entry*): M attempts to read a new symbol from Σ from the right-hand pushdown store. It pops that symbol from this right-hand store and pushes it onto the top of the left-hand store. If M is able to read such a symbol, then it performs the SEARCH operation. If M is not able to read such a symbol—i.e., M encounters the (right-hand) bottom symbol—then M halts (loop end and exit).

[2] In fact, any order on Σ^* suffices.

(ii) **SEARCH**: M reads the top t symbols from the left-hand store. It determines whether there exists a string u stored on the top $|u|$ positions of the left-hand store such that there exists v with $(u, v) \in R'$. If at least one such u exists, M chooses the longest u, remembers (u, v) and performs the REWRITE operation; in this case, we say that SEARCH "succeeds". Otherwise, SEARCH "fails"; then M restores the top t symbols of the left-hand store (loop end) and performs the READ operation again.

(iii) **REWRITE**: Since M has remembered (u, v), it pops the string u from the top of the left-hand store and pushes the string v onto the right-hand stack so that the leftmost symbol of v is on top of this store (*loop end*). Then M performs the READ operation again.

As defined for DTPDAs, the initial configuration will be $\perp q_0 w \perp$. Then M starts with the READ operation of the loop. It is easy to see that M satisfies the invariant mentioned above. For any configuration $\perp x_1 q x_2 \perp$ then x_1 is reducible if and only if there are $x_1', u \in \Sigma^*$ such that (i) $x_1 = x_1' u$ and (ii) all proper prefixes of x_1 are irreducible. Together with the fact that the SEARCH operation searches the longest such u this ensures that M simulates leftmost reductions. Since R' is terminating, M's computation must halt, with a configuration $\perp w' q \perp$ with $q \in Q, w' \in \textsc{Irr}(R')$ and $w \xrightarrow[R']{\text{lm}}{}^* w'$. Since $\textsc{Irr}(R') = \textsc{Irr}(R)$ and $R' \subseteq R$ this leads to the desired result: $w \xrightarrow[R]{\text{lm}}{}^* w' \in \textsc{Irr}(R)$. □

When we refer to the computation which is done by such an automaton M, we call it REDUCE-algorithm or -function. Formally, we use the following

Definition 2.23. *Let R be a terminating rewriting system and M a DTPDA as in the above proof. Then the operation of M that computes $w' \in \Sigma^*$ from input $w \in \Sigma^*$ with $w \xrightarrow[R]{\text{lm}}{}^* w' \in \textsc{Irr}(R)$ is called* REDUCE, *and we write:*

$$\textsc{Reduce}_R(w) = w' \text{ or simply } \textsc{Reduce}(w) = w'.$$

Remark 2.8. Because the REDUCE-algorithm is deterministic, it defines *unique leftmost normal forms* for each $w \in \Sigma^*$.

As can be seen the *basic operations* of the REDUCE-algorithm are:

1. For preprocessing (computing R') we need time c_0, which is not depending on the input $|w|$.
2. reading a single symbol from the top of the right-hand store and pushing it onto the left-hand store (READ operation, needing time c_1, not depending on $|w|$),
3. comparing two strings of maximum length $\max\{|u| \mid u \in \mathrm{dom}(R')\}$ (SEARCH operation, needing time c_2, not depending on $|w|$), and
4. pushing strings of maximum length $\mu_r = \max\{|v| \mid v \in \mathrm{range}(R')\}$ onto the right-hand store (REWRITE operation, needing time c_3, not depending on $|w|$).

Definition 2.24. *A rewriting system R on Σ is called* length-reducing *if for each $(u, v) \in R$ the length of the right-hand side is strictly smaller than that of the left-hand side, $|u| > |v|$.*

Clearly, a length-reducing rewriting system is Noetherian. If R is length-reducing, the number of basic operations needed to process an input with the REDUCE-algorithm can be computed as follows:

1. Each letter of the input has to be read, and after that we perform a SEARCH, for this we need time $(c_1 + c_2)|w|$.
2. Because a REWRITE reduces the length of the configuration by at least one, we need at most $|w|$ REWRITE operations, altogether needing time up to $c_3|w|$.
3. In the worst case, every REWRITE makes up to μ_r additional READ and SEARCH operations necessary. The time for these additional operations adds up to at most $\mu_r(c_1+c_2)|w|$. Since μ_r is fixed we get:
4. In the worst case, the REDUCE algorithm needs time

$$O(c_0 + ((\mu_r + 1)(c_1 + c_2) + c_3)|w|) = O(|w|).$$

Theorem 2.4. *Let R be a fixed length-reducing rewriting system on Σ. The following problem is solvable in linear time, depending on the length of the input:*

Instance: *a word $w \in \Sigma^*$.*

Result: *an irreducible string w' such that $w \xrightarrow{\mathrm{lm}}{}^* w'$.*

Proof. (given above) See [BO93] for further details.

Definition 2.25. *We call a rewriting system R* convergent *if it is terminating and confluent.*

Theorem 2.5. *If R is a convergent rewriting system on Σ then for all $x \in \Sigma^*$ $[x]$ has a unique normal form [BO93, theorem 1.1.12].*

Subsystems and normalization

Since a rewriting system R may contain two different rules $(u, v_1), (u, v_2)$ with $v_1 \neq v_2$ some problems can arise. There are two ways of dealing with those rules: Using equivalent subsystems in which no two different rules with identical left-hand sides exist or using normalized systems (which has some further effects).

Definition 2.26. *Let R, S be two rewriting systems on the same alphabet. They are called* equivalent *if and only if $\underset{R}{\longleftrightarrow}^* = \underset{S}{\longleftrightarrow}^*$.*

Lemma 2.3. *Let R be a convergent rewriting system. Let $S \subseteq R$ be a subsystem of R such that for any word $u \in dom(R)$ there is a unique v such that $(u, v) \in S$. Then:*

(i) S is convergent (that is, terminating and confluent),
(ii) $\textsc{Irr}(S) = \textsc{Irr}(R)$, and
(iii) S is equivalent to R.

Note on proof. This is a generalization of lemma 3 in [BO81]. There the rules of R are assumed to be strictly length reducing, i.e. for all $(u, v) \in R : |u| > |v|$, but the proof only uses that R and its subsystem are terminating. Obviously, S is terminating. Showing that it is confluent and that (ii) and (iii) hold can be achieved in the same manner that [BO81] used.

Therefore, if R is a convergent rewriting system and R' is the rewriting system constructed for the automaton which computes REDUCE, then R and R' are equivalent. As a consequence REDUCE computes unique normal forms. In this case, it is justified to write:

$$\textsc{Reduce}(w) = [w].$$

The normalization approach discussed below does not expect R to be confluent. But if R is confluent, then the result of the normalization is confluent, too. First we introduce some necessary definitions and results.

Definition 2.27. *A rewriting system R on Σ is called* normalized *if the following conditions hold for each rule $(u, v) \in R$:*

(i) $u \in \textsc{Irr}(R \setminus \{(u, v)\})$ *and*
(ii) $v \in \textsc{Irr}(R)$.

That means, the right-hand side v of each rule is irreducible and the left-hand side u can be reduced only by the rule (u, v) itself.

Note that in a normalized system no two rules $(u, v_1), (u, v_2)$ with $v_1 \neq v_2$ can exist.

Definition 2.28. *Let $>$ be a binary relation on Σ^*. Then $>$ is*

(i) *a* strict partial ordering *if it is irreflexive, antisymmetric, and transitive,*
(ii) *a* linear ordering *if it is a strict partial ordering and, for all $x, y \in \Sigma^*$, either $x > y, x = y$, or $y > x$ holds,*
(iii) admissible *if for all $u, v, x, y \in \Sigma^*$ $u > v$ implies $xuy > xvy$.*

Definition 2.29. *Let $>$ be a strict partial ordering on Σ^*. Then it is called* well-founded *if there is no infinite chain of the form $x_0 > x_1 > x_2 > \ldots$ with $x_i \in \Sigma^*$ for $0 \leq i \in \mathbb{N}$. If $>$ is linear and well-founded, then it is called* well-ordering.

Definition 2.30. *Let R be a rewriting system on Σ and $>$ be an admissible well-founded strict partial ordering on Σ^*. We say that $>$ is* compatible *with R if for, each rule $(u, v) \in R$, $u > v$ holds.*

Theorem 2.6. *Let R be a rewriting system on Σ. Then the following two statements are equivalent:*

(a) the reduction relation $\underset{R}{\longrightarrow}$ *is Noetherian;*

(b) there exists an admissible well-founded partial ordering $>$ *on* Σ^* *such that* $u > v$ *holds for each rule* $(u, v) \in R$.

For a proof see [BO93, theorem 2.2.4].

Theorem 2.7. *There is an algorithm which solves the following problem:*

Instance: *an admissible well-ordering* $>$ *on* Σ^*,
a finite rewriting system R *on* Σ *which is compatible with* $>$.

Result: *a normalized rewriting system* R' *which is equivalent to* R *and compatible with* $>$.
Furthermore, if R *is confluent then so is* R'.

Since the algorithm itself and the proof are of no interest for our investigations, we will not present them. They can be found in full detail in [BO93, theorem 2.2.13].

Local confluence

Theorem 2.8. *Let* R *be a terminating rewriting system. Then* R *is confluent if and only if* R *is locally confluent [BO93, theorem 1.1.13].*

Theorem 2.9. *There is an algorithm to decide the following question:*

Instance: *a finite terminating rewriting system* R.

Question: *is* R *(locally) confluent?*

Since the algorithm which decides this question reveals some ideas for constructing confluent rewriting systems, we discuss the full proof here.

First, remember that local confluence is a property of the complete rewriting system, not just of single rules or rule pairs.[3] Furthermore, all possible words from Σ^* are involved: For all $w, x, y \in \Sigma^*$ with $w \to x$ and $w \to y$ we have to check that there is a z such that $x \to^* z$ and $y \to^* z$. Obviously, one cannot test all possible ws algorithmically. The following argument shows that it is not necessary to do this.

[3] This shows that the term "local" is slightly misleading.

Proof. [BO81,BO93] Suppose $w \to x$ and $w \to y$. If $(u_1, v_1), (u_2, v_2) \in R$ and $w = w_1 u_1 w_2 u_2 w_3$. Then

$$w \underset{(u_1,v_1)}{\longrightarrow} w_1 v_1 w_2 u_2 w_3 \underset{(u_2,v_2)}{\longrightarrow} w_1 v_1 w_2 v_2 w_3$$

and also

$$w \underset{(u_2,v_2)}{\longrightarrow} w_1 u_1 w_2 v_2 w_3 \underset{(u_1,v_1)}{\longrightarrow} w_1 v_1 w_2 v_2 w_3.$$

As the result is $w_1 v_1 w_2 v_2 w_3$ in both cases, one does not need to test anything. Thus, one only has to test those cases were $w = w_1 u_1 w_2$ and $w = w_3 u_2 w_4$ with

(a) $|w_1 u_1| < |w_3 u_2|$ and $|w_2| < |u_2 w_4|$ or
(b) $|w_3 u_2| < |w_1 u_1|$ and $|w_4| < |u_1 w_2|$.

These cases occur when for some $x, y \in \Sigma^*$, $xy \neq \Box$, either $u_1 x = y u_2$ and $|x| < |u_2|$, or $u_1 = x u_2 y$. Therefore, to determine whether R is locally confluent, it is sufficient to consider pairs of rules from R.

Definition 2.31. *For each pair of (not necessarily distinct) rewriting rules* $(u_1, v_1), (u_2, v_2) \in R$ *let the set of* critical pairs *be:*

$$\begin{aligned} &\{[xv_1, v_2 y] \mid \text{there are } x, y \in \Sigma^*, xu_1 = u_2 y \text{ and } 0 < |x| < |u_2|\} \\ \cup\, &\{[v_1, xv_2 y] \mid \text{there are } x, y \in \Sigma^*, u_1 = x u_2 y\}. \end{aligned}$$

Because $xy = \Box$ is possible in the second set, rules $(u, v_1), (u, v_2) \in R$ with $v_1 \neq v_2$ lead to a critical pair, too.

Definition 2.32. *A critical pair* $[z_1, z_2]$ resolves *if and only if* z_1 *and* z_2 *have a common descendant.*

If a critical pair xv_1 and $v_2 y$ (or v_1 and $xv_2 y$) has two different irreducible descendants, then R is not confluent (hence, not locally confluent), because this would imply that the congruence class of $xu_1 = u_2 y$ (or $u_1 = x u_2 y$) has more than one irreducible element. On the other

hand, if every critical pair resolves, then the argument above implies that R is locally confluent.

To check whether a given critical pair resolves, we can use the algorithm REDUCE:

If R is confluent then all words have unique normal forms, therefore it is allowed to use leftmost reductions. On the other hand, assume R is not confluent. Then it does not matter when a critical pair does not resolve because the restriction to leftmost reductions "drops" some normal forms: Even if the congruence class was preserved by leftmost reductions, the critical pair would not resolve. In consequence it suffices to find a critical pair which is not resolving with leftmost reductions, and such a pair exists if and only if the rewriting system is not confluent.

So R is locally confluent if and only if for every critical pair $[z_1, z_2]$, $\text{REDUCE}(z_1) = \text{REDUCE}(z_2)$.

□

For the question of the complexity of testing local confluence with this algorithm, we refer the reader to [BO93]. A more efficient algorithm is described in [KKMN85].

Definition 2.33. *We call this algorithm* CONFTEST.

2.4 Growing context-sensitive languages

Chomsky introduced the notion of *generative grammars* to define languages [Cho59]:

Definition 2.34. *A* generative grammar *(only* grammar *for short) is a 4-tuple* $G = (\Sigma, \Gamma, S, P)$.

Γ *is the* alphabet of G,
$\Sigma \subset \Gamma$ *is the* terminal alphabet
$\Gamma \setminus \Sigma$ *are the* nonterminals of G.
$S \in \Gamma \setminus \Sigma$ *is the* start symbol.
P *is the set of* productions of G. *It is a rewriting system where each* $u \in dom(P)$ *contains at least one nonterminal.*

For distinction from rewriting systems we call a rule $(u, v) \in P$ *a production and denote it with* $(u \to v)$*. The* language defined by G *is*

$$L_G := \{w \in \Sigma^* \mid S \underset{P}{\longrightarrow}^* w\}.$$

A grammar G *is called* context-sensitive*, if each production* $r \in P$ *is of the form* $r = (uAw, uvw)$ *with* $u, v, w \in \Gamma^*$*,* $A \in \Gamma \setminus \Sigma$*.*

The class of languages generated by context-sensitive grammars is denoted by CSL. If a language L *is an element of CSL we also say* L is a CSL*. Furthermore, when speaking of more than one CSL, we use the plural "s", e.g. "*L_1 *and* L_2 *are CSLs."*[4]

A grammar is called (strictly) monotone *if the start symbol* S *does not appear on the right-hand side of any production in* P *and for each* $(u, v) \in P$*,* $|u| \leq |v|$ $(|u| < |v|)$*.*

Theorem 2.10. *The class CSL is characterized by monotone grammars [Cho59].*

Dahlhaus and Warmuth introduced the name *growing context-sensitive languages (GCSL)*:

Definition 2.35. *[DW86] The class of languages generated by strictly monotone grammars is called GCSL.*

They also proved that the word problem ($w \in L(G)$? for $w \in \Sigma^*$) for GCSL can be solved in polynomial time for a fixed grammar [DW86].

Definition 2.36. *A grammar* $G = (\Sigma, \Gamma, S, P)$ *is* weight-increasing *if there exists a weight function* $\varphi : \Gamma^* \to \mathbb{N}$ *such that for all* $(u \to v) \in P$ *the weight of* u *is less than that of* v*:* $\varphi(u) < \varphi(v)$*.*

Buntrock and Loryś achieved the following characterization:

Theorem 2.11. *The class GCSL is characterized by weight-increasing grammars [BL92].*

As Buntrock showed, GCSL are characterized by sTPDAs:

Theorem 2.12. *A language L is a GCSL if and only if it is accepted by a sTPDA A [Bun96].*

This justifies the definition of a deterministic variant of GCSL:

Definition 2.37. *The class of GCSL languages that are accepted by a deterministic sDTPDA is called deterministic GCSL, short DGCSL.*

It should be emphasized that neither the definition of GCSL by strictly monotone grammars nor the characterization by weight-increasing grammars require the grammars to be context-sensitive in the sense of definition 2.34. This reflects the fact that the name GCSL is coined for historical reasons, because for CSL monotone and context-sensitive grammars are equally powerful. A more detailed discussion of this can be found in [NW01a].

Closure properties of GCSL can be found in [BL92] and [BL94]. Buntrock [Bun96] also discusses *(growing) acyclic context-sensitive grammars*, (G)ACSL, defined under the name 1_A-grammars (1_B-grammars) by Parikh [Par66]. GACSL also have been investigated by Brandenburg [Bra74]. For those who are interested in the history and properties of this language class, we also mention [Gla64], [Boo69], and [BO98]. The position of (D)GCSL relative to the Chomsky hierarchy is discussed in [McN99], and in [BHNO00a] (see also [BHNO00b]) they are discussed in the context of McNaughton languages.

2.5 Church-Rosser languages

The definition of Church-Rosser languages was given by McNaughton, Narendran, and Otto [MNO88].

Definition 2.38. *A language $L \subseteq \Sigma^*$ is a* Church-Rosser language (CRL) *if there are*

- *an alphabet Γ,*

[4] The same convention will be used for all further language classes without explicit mentioning it.

- *a confluent and length-reducing rewriting R system on Γ,*
- $\Sigma \subset \Gamma$,
- *two words* $t_l, t_r \in (\Gamma \setminus \Sigma)^* \cap \textsc{Irr}(R)$,
- *a symbol* $Y \in (\Gamma \setminus \Sigma) \cap \textsc{Irr}(R)$
- *such that for all* $w \in \Sigma^*$, $t_l w t_r \underset{R}{\longrightarrow}^* Y$ *if and only if* $w \in L$.

R is a defining system *for the Church-Rosser language L.*

Furthermore, L is a Church-Rosser-decidable language (CRDL) *if additionally there is another symbol* $N \in (\Gamma \setminus \Sigma) \cap \textsc{Irr}(R)$ *such that* $t_l w t_r \underset{R}{\longrightarrow}^* N$ *if and only if* $w \notin L$.

Definition 2.39. *The* word problem for a fixed CRL L *is defined as follows:*

Let alphabets Γ and $\Sigma \subset \Gamma$, a confluent, length-reducing rewriting system R on Γ, $t_l, t_r \in \textsc{Irr}(R)$, and $Y \in (\Gamma \setminus \Sigma) \cap \textsc{Irr}(R)$, defining the CRL $L \subset \Sigma^*$, be given.

Instance: *A word $w \in \Sigma^*$.*

Question: *Is $w \in L$?*

Theorem 2.13. *Let L be a CRL. Then the word problem of L is decidable in linear time depending on the length of w.*

Proof. The problem can be solved with the Reduce algorithm, which works in linear time. See [MNO88] for details.

Definition 2.40. *A rewriting system $R \subseteq \Sigma^* \times \Sigma^*$ is called* weight-reducing, *if there exists a weight function $\varphi : \Gamma^* \to \mathbb{N}$ such that for all $(u, v) \in R$ the weight of v is strictly less than the weight of u: $\varphi(u) > \varphi(v)$.*

Niemann and Otto proved the following results which we summarize in three theorems [NO98].

Theorem 2.14. *A language $L \subseteq \Sigma^*$ is a CRL if and only if there are*

- *an alphabet Γ,*

- *a confluent and weight-reducing rewriting R system on Γ,*
- $\Sigma \subset \Gamma$,
- *two words* $t_l, t_r \in (\Gamma \setminus \Sigma)^* \cap \textsc{Irr}(R)$,
- *a symbol* $Y \in (\Gamma \setminus \Sigma) \cap \textsc{Irr}(R)$
- *such that, for all* $w \in \Sigma^*$, $t_l w t_r \underset{R}{\longrightarrow}^* Y$ *if and only if* $w \in L$.

In [BO98], the class of languages which are defined as in theorem 2.14 were called *generalized Church-Rosser languages* (GCRL). It was an open question whether GCRL = CRL. Buntrock and Otto showed that GCRL are characterized by sDTPDAs. Since Niemann and Otto showed that GCRL = CRL the following result can be obtained:

Theorem 2.15. *A language $L \subseteq \Sigma^*$ is a CRL if and only if there exists an sDTPDA M with $L(M) = L$. Therefore, the class CRL coincides with DGCSL (and with GCRL) [NO98].*

Theorem 2.16. *Let L be a CRL. Then L is a Church-Rosser-decidable language, too. That means: CRDL = CRL [NO98].*

The following closure properties (we assume the definitions to be common knowledge, they can be found in [Har78] or other textbooks about formal languages) hold:

Theorem 2.17. *The class of Church-Rosser languages is closed under the following operations:*

(a) Complementation,
(b) intersection with regular languages, and
(c) inverse homomorphism.

The class of Church-Rosser languages is not closed under the following operations:

(d) union of two (or more) CRLs,
(e) intersection of two (or more) CRLs,
(f) concatenation of two (or more) CRLs,
(g) ε-free homomorphisms, and
(h) arbitrary homomorphisms.

This overview of results is taken from [BHNO00a].

Given an LR(1)-Grammar G and a standard shift-reduce parser M accepting $L(G)$, one can simulate the behavior of M by an sDTPDA. This leads to:

Theorem 2.18. *The language class of the deterministic context-free languages* DCFL, see [Knu65]) *is included in the class of Church-Rosser languages: DCFL $\subset$ CRL. Note that this inclusion is proper [MNO88].*

Because of theorem 2.15 it is not necessary to use LR(1) grammars in Greibach normal form to establish this result, in contrast to the original proof in [MNO88, theorem 2.2.]. Proper inclusion follows from theorem 2.16 and [MNO88, theorem 2.4.].

Inclusion of DCFL in CRL is discussed in more detail in the application chapter, beginning on page 121.

Finally, we introduce a definition for systems that define Church-Rosser languages which is more in the style of grammars. This is equivalent to definition 2.38 because of theorem 2.14. Its main purposes are on the one hand to combine all parts necessary for a CRL-definition in one tuple and on the other hand to allow a shorter way of referencing the systems.

Definition 2.41. *A* Church-Rosser language system (CRLS) *is a 6-tuple $C = (\Gamma, \Sigma, R, t_l, t_r, Y)$ with*

- *finite alphabet Γ,*
- terminal alphabet $\Sigma \subset \Gamma$,
- *$\Gamma \setminus \Sigma$ is the alphabet of* nonterminals,
- *finite confluent weight-reducing rewriting system $R \subseteq \Gamma^* \times \Gamma^*$,*
- left and right end marker *words $t_l, t_r \in (\Gamma \setminus \Sigma)^* \cap \textsc{Irr}(R)$, and*
- accepting letter $Y \in (\Gamma \setminus \Sigma) \cap \textsc{Irr}(R)$.

The language defined by C *is:* $L_C := \{w \in \Sigma^* \mid t_l \cdot w \cdot t_r \rightarrow^*{}_R Y\}$.

Instead of $\underset{R}{\longrightarrow}$ *and* $\underset{R}{\longrightarrow}^*$ *we also use* $\underset{C}{\longrightarrow}$ *and* $\underset{C}{\longrightarrow}^*$ *when we speak about reductions used in CRLSs.*

Sometime we refer to R using $\textsc{Rules}(C) := R$.

Remark 2.9. There is some ambiguity of the term "Church-Rosser language system". It could not only be interpreted as a system defining *one* CRL but also as *a system (i.e., a set) of some CRLs.* Since we do not use the second notion in our investigations, this should not be a problem.

Considering the rather short time since the first definition of CRL, there are a lot of results concerning them. Their relation to GCSL has been discussed in [BO98] and [NO98]. Concerning other language classes and automata models connected to CRL, important references are [NO99], [Nie00], [McN99], and [BHNO00a].

Chapter 3

A Characterizing Theorem

As already noted in theorem 2.15, Niemann and Otto proved that CRL are the deterministic variant of GCSL [NO98]. The term "'growing *context-sensitive* language"' is somewhat misleading, since weight-increasing grammars are not context-sensitive in the sense of [Cho59]: they are only *monotone* grammars. The term is only used for historical reasons, it was coined by Dahlhaus and Warmuth [DW86]. In this chapter we give a characterization of CRL which, when transfered to GCSL, justifies to call them context-sensitive.

In the first section we give a definition for so-called *context-splittable* CRLSs. The proof that this is a normal form for CRLSs is rather complicated, so we split it into different sections. A short description of the construction principles is contained in the second section. The next two sections describe the basic necessary technicalities which are not central to the idea of the proof, but can be derived simply from the fact that sDTPDAs characterize CRL. We devote the fifth section to the main strategy of the proof, which we call *"weight-spreading"*. In order to give a complete proof, a rather number of cases has to be considered. An overview of these cases is given in the sixth section, and the seventh section contains an example case. We have to make some efforts to make the resulting rewriting system confluent. This is discussed in the eighth section. Weight reduction is another property that has to be considered. For the complicated part of the construction—the weight-spreading technique—we give a detailed examination in the ninth section. For completeness we have to construct final rules, this is done in the tenth section.

The result of these efforts leads to the eleventh section, where we show correctness of the construction. In the twelfth section we link our characterization results about CRL to GCSL. Additionally, we discuss the topics of derivation trees in this section. The last but one section of this

chapter contains necessary case distinctions which we extracted from the proof for the purpose of better readability.

We conclude this chapter with a section in which we shortly discuss why this result is not simply a normal form, but a characterization theorem.

Early states of these research results have been published in a technical report [Woi00a], and an extended abstract was presented in [Woi01], which is improved here especially w.r.t. the question of confluence.

3.1 The normal form

Definition 3.1. *A CRLS* $C = (\Gamma, \Sigma, R, ¢, \$, Y)$ *is* context-splittable (C is a csCRLS) *if*

$$¢, \$, Y \in \textsc{Irr}(R) \cap \Gamma \setminus \Sigma$$

(let the inner alphabet*be* $\Gamma_{inner} := \Gamma \setminus \{¢, \$, Y\}$*) and for any rule* $r \in R$ *there exists a splitting* (u, v, w, x) *with:*

1. $r = (uvw, uxw)$.
2. *v is non-empty:* $v \in \Gamma^+$.
3. *uvw may contain at most one ¢ and if so at its beginning. Also it may contain at most one \$ which only may appear at the end. All other letters of uvw have to be from the inner alphabet Γ_{inner}:*

$$uvw \in \{¢, \square\} \cdot \Gamma^*_{inner} \cdot \{\$, \square\}.$$

4. *x is a single letter not equal to ¢ or \$ or it is the empty word:*

$$x \in \Gamma_{inner} \cup \{Y, \square\}.$$

5. *If v contains ¢ or \$, then $x = Y$, u and w are empty, and v begins with ¢ and ends with \$:*

$$v \in ¢ \cdot \Gamma^*_{inner} \cup \Gamma^*_{inner} \cdot \$ \Longrightarrow v \in ¢ \cdot \Gamma^*_{inner} \cdot \$ \wedge u = w = \square \wedge x = Y.$$

6. *If $x = Y$, then u and w are empty, and v begins with ¢ and ends with \$:*

$$x = Y \Longrightarrow v \in ¢ \cdot \Gamma^*_{inner} \cdot \$ \wedge u = w = \square.$$

The splitting (u, v, w, x) *of a rule* r *allowed by this definition is called a* context-splitting*,* u *and* w *are called the* left and right context*.*

Remark 3.1. It is obvious that $x = \square$ is only necessary if $u \in \{\square, ¢\}$ and $w \in \{\square, \$\}$ cannot be avoided, since otherwise one could transfer a letter of u or w to x.

Example 3.1. These are some examples for the meaning of the definition (the splittings are marked by dots):

- $ab \cdot dea \cdot ab \to ab \cdot a \cdot ab$,
- $ab \cdot de \cdot aab \to ab \cdot \square \cdot aab$ is another splitting of the same rule,
- $¢ \cdot ab \cdot \$ \to ¢ \cdot \square \cdot \$$,
- $¢ \cdot ab\$ \cdot \square \to ¢ \cdot \$ \cdot \square$ is *not a valid* splitting,
- $abc \to \square$ is a *deleting* context-splittable rule, and
- $abcd \to dcba$ is a rule that is not context-splittable.

Remark 3.2. We want to distinguish the notion of context-splittable CRLSs from context-sensitive grammars, because the former uses reductions and the latter productions for defining languages. Especially deleting rules of the form $v \to \square$ have no counterpart in context-sensitive grammars. Therefore we do not use the term context-sensitive. Note that all other deleting rules with $u \notin \{\square, ¢\}$ or $w \notin \{\square, \$\}$ can be split in a different way such that x is not the empty word, just as in the example given above.

Theorem 3.1. *Let* $C = (\Gamma, \Sigma, R, t_l, t_r, Y)$ *be a CRLS with language* L_C*. Then there exists a csCRLS* C' *with* $L_{C'} = L_C$*.*

Proof. We will give an effective construction for such a new csCRLS C'.

Remark 3.3. We already have a formal definition of the automata for the language family CRL, the sDTPDAs. Part of our construction will be the simulation of such an automaton. We refer the reader to the proof of theorem 2.3 as a guideline to the construction of an sDTPDA from a CRLS, which basically consists of a READ/SEARCH/REWRITE loop. In the following, we will call the SEARCH part a *shift operation* and the REWRITE part a *reduce operation.*

Note that since after a reduction operation the left-hand stack will always be irreducible, one can directly combine one shift with each reduce operation, as in [MNO88].

3.2 The construction principles

Without restriction of generality we may assume that R is length reducing [NO98]. Furthermore, we can assume that R is normalized: In a confluent rewriting system, rules that have a left-hand side which can be reduced by another rule can be dropped, and reducing right-hand sides does not cause a conflict with length-reduction.

In order to construct a csCRLS C', we will use the following four principles:

1. Analogous to the automaton model our new system will have the property that during the whole reduction process there is always exactly one place in the word where the next reduction rule can be applied.
2. We will use a compression alphabet which can store more than one letter of the input (respectively, the derived words) in one letter. This information will be represented by subscripts of the compression letters.
3. These compression letters will be enriched by surplus letters in their subscripts in order to spread necessary weight reductions over more than one letter.
4. Rules of the original system will, in most cases, be simulated by three or four rules in the new system.

The confluent weight reducing system will be built of five parts R_1 to R_5.

Definition 3.2. *With Γ being the alphabet of C which consists of all terminal and nonterminal letters, let $\overline{\Gamma}$ be a new alphabet, which is a disjoint copy of Γ. Then $^{-}$ denotes the bijective morphism that maps Γ into $\overline{\Gamma}$, e.g. a to $\overline{a}$. Let $\sharp$, $¢$, and $\$$ be new symbols. Let $\Gamma_\sharp := \Gamma \cup \{\sharp\}$ and $\overline{\Gamma}_\sharp := \overline{\Gamma} \cup \{\sharp\}$. Define*

$$W_\sharp := \overline{\Gamma}_\sharp^* \cdot \Gamma_\sharp^* \cap ((\sharp^{\leq 2} \cdot ((\overline{\Gamma} \cup \Gamma) \cdot \sharp\sharp)^* \cdot (\overline{\Gamma} \cup \Gamma) \cdot \sharp^{\leq 2}) \cup \sharp^{\leq 2}),$$

where $\sharp^{\leq 2}$ *is a shorthand for* $\{\square, \sharp, \sharp\sharp\}$.

As can be seen, this regular language consists of words, where between letters of Γ or $\overline{\Gamma}$ there are always exactly two $\sharp$'s. This language will not only be used to define the compression alphabet, but also to make some definitions during the construction process easier. The purpose of the $\sharp$'s will be explained later.

Definition 3.3. *Let* $\mu_l = \max\{|u| \mid u \in dom(R)\}$ *and* $\mu_r = \max\{|v| \mid v \in range(R)\}$ *be the maximum length of the respective rule sides. Because* R *is length reducing* $\mu_l > \mu_r$ *holds. Let* $\mu := \max\{\mu_l, |t_l|, |t_r|\}$. *The* compression alphabet Γ_1 *is defined as:*

$$\Gamma_1 := \{\xi_w \mid w \in W_\sharp \wedge 1 \leq |w| \leq 3\mu + 5\}.$$

The elements of Γ_1 *are called* compression letters. *For distinction, letters in the index of a compression letter will be called* index letters.

Remark 3.4. At least $\mu + 1$ letters of the original alphabet Γ can be stored in one compression letter.

If we need to delete the surplus $\sharp$'s, we use the following morphism:

Definition 3.4. *Let* $\hat{a}$ *be the morphism* $\hat{}: (\Gamma_\sharp \cup \overline{\Gamma}_\sharp \cup \{¢, \$\})^* \to (\Gamma \cup \overline{\Gamma} \cup \{¢, \$\})^*$ *defined by:*

$$\hat{x} := \begin{cases} \square & x = \sharp \\ x & \text{else.} \end{cases}$$

Sometimes it is necessary to extract the information of the subscripts in a word from Γ_1^*:

Definition 3.5. *Let* $\check{}$ *be the morphism* $\check{}: (\Gamma_1 \cup \overline{\Gamma} \cup \Gamma \cup \{¢, \$\})^* \to (\overline{\Gamma}_\sharp \cup \Gamma_\sharp)^*$ *defined by:*

$$\check{x} := \begin{cases} w & x = \xi_w \in \Gamma_1 \\ x & \text{else.} \end{cases}$$

We assume, without loss of generality, that brackets are not in our alphabets so far and use them in the following for better readability.

3.3 Translating the input

The first step in the simulation of C is to translate the input into the compression alphabet. At the same time, we will take care of t_l and t_r. The new end marker letters (not words!) of C' will be $t'_l := ¢$ and $t'_r := \$$. Short words $w \in L_C$, $|w| \leq 2$ will be handled separately, they are directly added to the language $L_{C'}$. For them, a set R_1 of rules is used:

$$R_1 := \{(¢w\$, Y) \mid w \in L_C \wedge |w| \leq 2\}.$$

Obviously, R_1 can be computed easily.

For the translating rule system we will give a new set of rules R_2, but first we have a look at the specification of it. Decompose t_l and t_r into single letters in the following way: $t_l = a_1 a_2 \cdots a_{|t_l|}$ and $t_r = c_1 c_2 \cdots c_{|t_r|}$. R_2 will be designed to be a confluent and weight reducing rewriting system such that for every $w \in \Sigma^{\geq 2}$ with $w = b_1 b_2 \cdots b_i \cdots b_{|w|}$, $b_i \in \Sigma (1 \leq i \leq |w|)$ we can make the following reduction with R_2:

$$¢w\$ \xrightarrow[R_2]{*} ¢\xi_{(\overline{a}_1\sharp\sharp)(\overline{a}_2\sharp\sharp)\cdots(\overline{a}_{|t_l|}\sharp\sharp)(\overline{b}_1\sharp\sharp)}\xi_{b_2\sharp\sharp}\xi_{b_3\sharp\sharp}\cdots\xi_{b_{|w|-1}\sharp\sharp}\xi_{(b_{|w|}\sharp\sharp)(c_1\sharp\sharp)(c_2\sharp\sharp)\cdots(c_{|t_r|}\sharp\sharp)}\$,$$

where we ensure that R_2 does not do more than such translations. Especially, the right-hand sides of the reductions given above are required to be irreducible in R_2. Furthermore, we require that the first translated letter after the ¢ always appears in the last reduction step. This is necessary to give a precise moment in the reduction after which the rule sets defined in the following parts of the construction can begin to work.

After this explanation, the following definition of R_2 should be clear. Assume t_l and t_r to be composed of letters a_i and c_i as above. R_2 will be composed of the two parts $R_{2,1}$ and $R_{2,2}$. We start with the translation from the right with the system $R_{2,1}$:

$$\begin{aligned} R_{2,1} := \quad & \{(def\$, de\xi_{(f\sharp\sharp)(c_1\sharp\sharp)(c_2\sharp\sharp)\cdots(c_{|t_r|\sharp\sharp})}\$) \mid d, e, f \in \Sigma\} \\ & \cup \{(de\xi_{(f\sharp\sharp)(c_1\sharp\sharp)\cdots(c_{|t_r|\sharp\sharp})}\$, d\xi_{e\sharp\sharp}\xi_{(f\sharp\sharp)(c_1\sharp\sharp)\cdots(c_{|t_r|\sharp\sharp})}\$) \mid d, e, f \in \Sigma\} \\ & \cup \{(de\xi_{f\sharp\sharp}, d\xi_{e\sharp\sharp}\xi_{f\sharp\sharp} \mid d, e, f \in \Sigma\} \end{aligned}$$

Now, the translation has to be finished:

$$R_{2,2} := \{(¢e\xi_{f\sharp\sharp}, ¢\xi_{(\overline{a}_1\sharp\sharp)(\overline{a}_2\sharp\sharp)\cdots(\overline{a}_{|t_l|\sharp\sharp})(\overline{e}\sharp\sharp)}\xi_{f\sharp\sharp}) \mid e, f \in \Sigma\}$$

Note that if $t_l = \square$ then the over-lined e is the first index letter after the translation.

Finally, $R_2 := R_{2,1} \cup R_{2,2}$.

Example 3.2. Let $t_l = dd, t_r = c$, and $w = abad$. Then R_2 translates the input to:

$$¢\xi_{\overline{d}\sharp\sharp\overline{d}\sharp\sharp\overline{a}\sharp\sharp}\xi_{b\sharp\sharp}\xi_{a\sharp\sharp}\xi_{d\sharp\sharp c\sharp\sharp}\$$$

The following can be easily verified:

Claim. $R_1 \cup R_2$ is confluent and, with a suitable weight function, weight reducing (since each of these rules translates exactly one terminal into a nonterminal, we simply have to assign weight big enough to the terminals). The last rule applied from $R_1 \cup R_2$ is either an accepting rule from R_1 or a rule from $R_{2,2}$. Thus, the first time an over-lined letter appears in the process is after the complete input has been translated to the compression alphabet.

The rightmost over-lined letter marks the position at which the simulation of C will work with the following rule sets. In general, the rightmost over-lined index letter of a compressed word can be identified with the head position of the automaton described above.

The $\sharp$'s are the *surplus letters* mentioned in the list of construction principles. Moving them to the left or right in a suitable manner will be necessary for the weight reduction property of the rules to follow.

3.4 Simulating shift operations

The next step is similar to the shift operations of automata for CRLs. Sometimes it is necessary to *move right* (that is, shift) the position of a possible next reduction.

Definition 3.6. *Given the rewriting system R, we will call a word w a* shift suffix*, if it is not a suffix of the left-hand side of a rule of R.*

Whenever the automaton accepting the language of C finds a shift suffix on the top of its left-hand stack, it must perform a shift operation.

Definition 3.7. *Given $R \in \Gamma^+ \times \Gamma^*$, a* sufficient set of shift suffixes *is a set $S \subseteq \Gamma^+$ of shift suffixes satisfying:*

$$\begin{aligned}\forall w \in \textsc{Irr}(R) \cap \Gamma^+ : (\quad & (\exists u \in \mathit{dom}(R) : w \in \mathit{Suff}(u)) \\ & \vee (\exists w' \in \Gamma^*, u \in \Gamma^+ : w = w'u \wedge u \in S)).\end{aligned}$$

Definition 3.8. *The following set of shift suffixes will be denoted with S:*

$$\begin{aligned}S := \{w \mid\ & w \in \Gamma^{\leq \mu_l} \\ & \wedge \forall u \in \mathit{dom}(R) : w \notin \mathit{Suff}(u) \\ & \wedge (\forall w' \in \mathit{Suff}(w) \setminus \{w\} : (\exists u' \in \mathit{dom}(R) : w' \in \mathit{Suff}(u')))\}\end{aligned}$$

Accordingly, $\overline{S}$ is the picture of S under $^{-}$.

The next two propositions should be intuitive. We provide proofs for them, anyhow.

Proposition 3.1. *S is a sufficient set of shift suffixes.*

Proof. From the first two conditions it follows that all elements of S are shift suffixes. The last condition excludes only words from S that are too short to be shift suffixes. Assume $w \in \textsc{Irr}(R) \cap \Gamma^+$.

If there exists $u \in \mathrm{dom}(R)$ such that $w \in \mathrm{Suff}(u)$, the first condition of definition 3.8 holds and neither w nor a suffix of w can be a shift suffix.

Otherwise, there must exist $w' \in \Gamma^*$ and u with $|u| \leq \mu_l$ such that $w = w'u$ and u is a shift suffix. Assume u is the shortest such shift suffix, then it is an element of S. □

Proposition 3.2. *S is minimal: Any sufficient set of shift-suffixes S' is a superset of S or equal to it.*

Proof. Assume $u \in S'$. There are two possibilities:

Case 1. There exists no shift suffix v such that $v \in \mathrm{Suff}(u)$. Then u is element of S, because S is sufficient.

Case 2. There exists a shift suffix v such that $v \in \mathrm{Suff}(u)$. Then there exists a shortest suffix v' of v which fulfills case 1. Therefore, $u, v \notin S$ and $v' \in S$. □

Since in our new systems all reductions have to take place *within the indices* and cannot directly work on the letters of the original alphabets we need to take care of this. Especially, any (sub)word of original letters can be distributed over more than one letter of the compression alphabet, and the $\sharp$'s have to be considered, too. So, we *translate* S in the following way:

Definition 3.9. *Define the S_ξ set of* translated shift suffixes *as:*

$$\begin{aligned} S_\xi := \{\xi_{w_1}\xi_{w_2}\cdots\xi_{w_{n-1}w_n} \mid\ & 2 \leq n \wedge w_i \in W_\sharp \cap \overline{\Gamma}_\sharp^+ \ (1 \leq i \leq n-2) \\ & \wedge\ w_{n-1} \in W_\sharp \cap \overline{\Gamma}_\sharp^* \\ & \wedge\ w_n \in W_\sharp \cap (\Gamma \cdot \Gamma_\sharp^*) \\ & \wedge\ w_1 w_2 \cdots w_n \in W_\sharp \\ & \wedge\ (\exists v_1, v_2 : v_1 v_2 = \hat{w}_1 \wedge v_2 \hat{w}_2 \cdots \hat{w}_{n-1} \in \overline{S}) \\ & \wedge\ \hat{w}_2 \cdots \hat{w}_{n-1} \notin \overline{S}\}. \end{aligned}$$

It is also possible that the simulated left-hand store does contain an irreducible word which is not a shift suffix. This happens when the word in the store is too short and a suffix of a left-hand-side of a rule. This case is handled with the following set.

Definition 3.10. *The set of* short shift words *is defined as:*

$$\begin{aligned} S_{¢} := \{¢\xi_{w_1}\xi_{w_2}\cdots\xi_{w_{n-1}w_n} \mid\ & 2 \le n \wedge w_i \in W_\sharp \cap \overline{\Gamma}_\sharp^+ \ (1 \le i \le n-2) \\ & \wedge\ w_{n-1} \in W_\sharp \cap \overline{\Gamma}_\sharp^* \\ & \wedge\ w_n \in W_\sharp \cap (\Gamma \cdot \Gamma_\sharp^*) \\ & \wedge\ w_1 w_2 \cdots w_n \in W_\sharp \\ & \wedge\ (\exists u \in dom(R), u' \in Suff(u) \cap \textsc{Irr}(R) : \\ & \quad \hat{w}_1\hat{w}_2\cdots\hat{w}_{n-1} = \overline{u'})\}. \end{aligned}$$

In order to get a confluent rewriting system in combination with the rules defined in the later sections, we have to use a kind of lookahead set. We will discuss its exact role in the following sections and simply give the definition here:

Definition 3.11. *The* lookahead set S_l *is defined as*

$$\begin{aligned} S_l := \ & \{\xi_{w_1}\xi_{w_2} | w_1 w_2 \in W_\sharp \wedge \hat{w}_1\hat{w}_2 \in \Gamma^*\} \\ & \cup\{\xi_{w_1\sharp\sharp}\$ | w_1\sharp\sharp \in W_\sharp \wedge \hat{w}_1 \in \Gamma^*\} \\ & \cup\{\$\}. \end{aligned}$$

Furthermore, we use the following two abbreviations:

Definition 3.12.

$$W_{\sharp,\$} := W_\sharp \cup ((W_\sharp \cap W_\sharp \cdot \{\sharp\sharp\}) \cup \{\sharp, \Box\}) \cdot \{\$\}.$$

That means, each word in $W_{\sharp,\$}$ is an arbitrary word from $W_\sharp$, or it is a word from $W_\sharp$ ending with two $\sharp$'s concatenated with \$, or it is the word $\sharp\$$, or it is simply a \$.

Definition 3.13. *The set* $W_{\sharp,\$,\Gamma}$ *is defined as the subset of* $W_{\sharp,\$}$ *whose words start with a letter from* Γ*:*

$$W_{\sharp,\$,\Gamma} := W_{\sharp,\$} \cap \Gamma \cdot W_\sharp \cdot \{\Box, \$\}.$$

Now we construct R_3: For each $w = \xi_{w_1}\xi_{w_2}\cdots\xi_{w_{n-2}}\xi_{w_{n-1}w_n}$ with $w \in S_\xi$ or $¢w \in S_{¢}$ and $w' \in S_l$ such that $\check{w}\check{w}' \in W_{\sharp,\$}$ we add a rule $r_{w,w'}$ to the new system R_3. Assume that $w_n = aw'_n$ with $a \in \Gamma$ and $w'_n \in W_\sharp$. If $w \in S_\xi$ the rule to be added for w and w' is

$$r_{w,w'} := (\xi_{w_1}\xi_{w_2}\cdots\xi_{w_{n-2}}\xi_{w_{n-1}w_n}w', \xi_{w_1}\xi_{w_2}\cdots\xi_{w_{n-2}}\xi_{w_{n-1}\overline{a}w'_n}w').$$

If $¢w \in S_¢$ we add the rule

$$r_{w,w'} := (¢\xi_{w_1}\xi_{w_2}\cdots\xi_{w_{n-2}}\xi_{w_{n-1}w_n}w', ¢\xi_{w_1}\xi_{w_2}\cdots\xi_{w_{n-2}}\xi_{w_{n-1}\overline{a}w'_n}w').$$

Notice that the cases $w \in S_\xi$ and $¢w \in S_¢$ are disjoint.

Remark 3.5. If a shift is necessary but no next letter without over-lining exists no next reduction is possible. Then the simulated system also would reduce to an irreducible word.

Claim. R_3 is confluent.

Proof. The left-hand sides of rules derived from the translated shift words in S_ξ have no nontrivial overlaps with the rules derived from short shift words in $S_¢$. Because S is minimal, left-hand sides of rules derived from S_ξ do not overlap with each other. Left-hand side of rules from $S_¢$ do not overlap because of the ¢. So, there are no nontrivial overlaps between left-hand rule sides in R_3. Therefore it is confluent. □

It is also possible to show that R_3 is weight reducing. The matter of weight reduction will be discussed later, at the moment simply assume that over-lined letters in the subscripts of compression letters add slightly less to the weight.

Claim. $R_1 \cup R_2 \cup R_3$ is confluent and weight reducing. There are no overlaps between the former two and R_3, because on the left-hand sides of R_1 and R_2 rules there are never over-lined letters.

Example 3.3. Assume $bba \in S$. Then $w = \xi_{\overline{b}\sharp\sharp\overline{b}\sharp\sharp\overline{a}\sharp}\xi_{\sharp c\sharp} \in S_\xi$. Assume $w' = \xi_\sharp\$$. This leads to the rule:

$$r_w = (\xi_{\overline{b}\sharp\sharp\overline{b}\sharp\sharp\overline{a}\sharp}\xi_{\sharp c\sharp}\xi_\sharp\$, \xi_{\overline{b}\sharp\sharp\overline{b}\sharp\sharp\overline{a}\sharp}\xi_{\sharp\overline{c}\sharp}\xi_\sharp\$) \in R_3.$$

3.5 The "weight-spreading" technique

Now we reach the core of the construction, which will be called *weight-spreading*. The main idea is to simulate rules of R piecewise. This simulation is similar to the construction of a context-sensitive grammar from a monotone one [Cho59]. In order to achieve a weight reducing system a second principle is used: the simulation will reduce the length of the subscripts of the compression letters.

In order to make the construction more understandable, we provide two examples. The first is *not working as desired*, it is used to illustrate why two $\sharp$'s are put between the letters of $\overline{\Gamma}$ or Γ, respectively. The second example shows the correct construction at work.

Example 3.4. Consider the rule $(aaaa, bbb)$ and assume we had used only a single $\sharp$ as surplus letter. The following possible reduction could be used (each step being identified with a rule):

$$\xi_{\overline{a}\sharp\overline{a}}\xi_{\sharp\overline{a}\sharp}\xi_{\overline{a}\sharp a}$$

$\longrightarrow$ 1. "lock" with new nonterminal

$$\xi_{\overline{a}\sharp\overline{a}}\xi_{\sharp\overline{a}\sharp}\xi_{t}$$

$\longrightarrow$ 2. change first letter

$$\xi_{\overline{b}\sharp}\xi_{\sharp\overline{a}\sharp}\xi_{t}$$

$\longrightarrow$ 3. change middle letter

$$\xi_{\overline{b}\sharp}\xi_{b\sharp}\xi_{t}$$

$\longrightarrow$ 4. change last letter, remove lock

$$\xi_{\overline{b}\sharp}\xi_{b\sharp}\xi_{b\sharp a}$$

Fig. 3.1: An incorrect simulation

In the first rewriting step we use the known shift position (the last overlined subscript letter) to assert that no other rules can interfere. During the next three steps we simulate the actual reduction. Note that the first

b also is over-lined. This is possible because after each reduction in the corresponding automaton a shift is necessary, before another reduction can take place.

This example shows the problems we have to deal with. A look at the last letter in the first line and the last letter in the last line reveals that the length of the subscript did not change. In consequence, if we would do this for all rules in any CRLS, this could cause weight reduction problems.

Example 3.5. Therefore, we now change the example insofar, as we introduce double $\sharp$'s. Let the rule in question again be $(aaaa, bbb)$:

$$\xi_{\overline{a}\sharp\sharp\overline{a}}\xi_{\sharp\sharp\overline{a}\sharp\sharp}\xi_{\overline{a}\sharp\sharp a}$$

$\longrightarrow$ 1. “lock” with new nonterminal

$$\xi_{\overline{a}\sharp\sharp\overline{a}}\xi_{\sharp\sharp\overline{a}\sharp\sharp}\xi_t$$

$\longrightarrow$ 2. change first letter

$$\xi_{\overline{b}\sharp\sharp}\xi_{\sharp\sharp\overline{a}\sharp\sharp}\xi_t$$

$\longrightarrow$ 3. change middle letter

$$\xi_{\overline{b}\sharp\sharp}\xi_{b\sharp\sharp b}\xi_t$$

$\longrightarrow$ 4. change last letter, remove lock

$$\xi_{\overline{b}\sharp\sharp}\xi_{b\sharp\sharp b}\xi_{\sharp\sharp a}$$

Fig. 3.2: A correct simulation

The $\sharp$'s are used to spread the length reduction of the original rule over the compression letters in the simulation. Observe that (especially) in the last step of the example the result contains three compression letters. More than three letters are not necessary during any rule simulation, because a right-hand side of a rule and the added $\sharp$'s fit into one compression letter (to be exact, sometimes some *unchanged contexts* appears on the right-hand side, necessitating four letters). Basically, the weight of a compression letter will be computed from the number of its index

letters (over-lined letters add *slightly* less to the weight). Therefore in the worst case—when the original rule has a length reduction of one—we reduce the number of index letters by three and spread this reduction over the resulting three compression letters. This is the cause for adding two $\sharp$'s after each index letter of $\Gamma \cup \overline{\Gamma}$ during the translation with R_2.

For generalizing this example, a lot of cases have to be handled. The main idea is to identify all possibilities how the left-hand side of an original rule can be split over one or more letters of the compression alphabet. Also, sometimes it is necessary to allow some unchanged context in the first compression letter. Furthermore, the question of finding the right place for the reduction to work has to be handled.

At this point we can explain the necessity of the lookahead set for our construction. Assume there is another rule whose left-hand side ends with a letter b. As soon as the middle letter is changed in our example (step 3), a simulation of that second rule could start, therefore possibly preventing the ξ_t from being removed for the rest of the simulation. This would result in a rewriting system which is not confluent. So our example only showed the simulation part of our construction, but not how the confluence is achieved. In the simulation rules which we are going to add, this is prevented by looking two compression letters further ahead whether there is a locking nonterminal. If there are no two compression letters, but only one and the end marker letter \$, or even only the \$, the same applies. How this works can be seen in detail in the following section.

3.6 Case distinctions

In this section we will give the distinction of main cases and an example how to handle one of these cases. With this example the reader should be able to understand the principles of the construction. The complete construction can be found in section 3.13.

In some cases, the simulation is very easy. Especially, when the complete reduction can be simulated within one letter of the compression alphabet. We handle this first case with the following set:

$$
\begin{aligned}
T_1 := \{(u, v, \xi_{w_1 w_2 w_3} w) \mid\ & (u, v) \in R \\
& \wedge\ w_1 w_2 w_3 \in W_\sharp \\
& \wedge\ w \in S_l \\
& \wedge\ w_1 \in \overline{\Gamma}_\sharp^* \\
& \wedge\ \hat{w}_2 = \overline{u} \wedge w_2 \in \overline{\Gamma} \cdot \overline{\Gamma}_\sharp^* \cdot \overline{\Gamma} \cdot \{\sharp\sharp\} \\
& \wedge\ w_3 \check{w} \in W_{\sharp,\$,\Gamma}\}.
\end{aligned}
$$

Now, for each element of $t \in T_1$ assume $u = a_1 \cdots a_{|u|}, a_i \in \Gamma$ $(1 \leq i \leq |u|)$ and $v = b_1 b_2 \cdots b_{|v|}, b_i \in \Gamma$ $(1 \leq i \leq |v|)$. For each t add a rule r_t to a new system R_4 :

$$
r_t := (\xi_{w_1 w_2 w_3} w, \xi_{w_1 (\overline{b}_1 \sharp\sharp)(b_2 \sharp\sharp) \cdots (b_{|v|} \sharp\sharp) w_3} w)
$$

If $w_1 = w_3 = v = \Box$, this would produce a letter $\xi_\Box$, which is not in the compression alphabet. In these cases identify $\xi_\Box \equiv \Box$. Then the rule will simply delete one symbol.

Now the difficult part of the construction will be discussed. What has to be done, if the left-hand side of the original rule is not in one letter but distributed over the indices of several compression letters? Again we use a set which contains all cases of possible rule applications that are not covered by the above set T_1:

$$
\begin{aligned}
T_2 := \{(u, v, \xi_{w_1 w_2} \xi_{w_3} \xi_{w_4} \cdots \xi_{w_{n-2}} \xi_{w_{n-1} w_n} w) \mid\ & (u, v) \in R \\
& \wedge\ 4 \leq n \\
& \wedge\ w_1 \cdots w_n \in W_\sharp \\
& \wedge\ w \in S_l \\
& \wedge\ w_1 \in \overline{\Gamma}_\sharp^* \\
& \wedge\ w_2 w_3 \cdots w_{n-1} \in \overline{\Gamma} \cdot \overline{\Gamma}_\sharp^* \cdot \overline{\Gamma} \cdot \{\sharp\sharp\} \\
& \wedge\ w_2 \neq \Box \\
& \wedge\ \hat{w}_2 \hat{w}_3 \cdots \hat{w}_{n-1} = \overline{u} \\
& \wedge\ w_n \check{w} \in W_{\sharp,\$,\Gamma}\}.
\end{aligned}
$$

For each $t \in T_2$ (T_2 is finite) a set of rules is added to R_4.

This also needs some further nonterminals. These will be collected in the set Γ_2, whose elements will be called *locking symbols*. Again, we assume $u = a_1 \cdots a_{|u|}, a_i \in \Gamma$ $(1 \leq i \leq |u|)$ and $v = b_1 b_2 \cdots b_{|v|}, b_i \in \Gamma$ $(1 \leq i \leq$

$|v|$). Whenever we speak of three or four rules these are a simulation of an original rule in as many steps.

The following ten cases have to be dealt with, some *having up to 23 sub-cases.* Let $t = (u, v, \xi_{w_1 w_2} \xi_{w_3} \xi_{w_4} \cdots \xi_{w_{n-2}} \xi_{w_{n-1} w_n} w) \in T_2$:

1. $n = 4, w_3 = \square$ (already covered by the rules for T_1, see page 62)
2. $n = 4, v = \square, w_3 \neq \square$ (all deleting rules need some extra care, four sub-cases)
3. $n = 4, v \neq \square, w_3 \in \{\sharp, \sharp\sharp\}$ (similar to the rules for T_1, no sub-cases, one rule for t)
4. $n = 4, v \neq \square, |w_3| > 2$ (five sub-cases, up to three rules for t)
5. $n = 5, v \neq \square, w_4 = \square$ (already captured by 3. and 4.)
6. $n = 5, v = \square, w_4 \neq \square$ (again deleting rule, four sub-cases)
7. $n = 5, v \neq \square, w_4 \in \{\sharp, \sharp\sharp\}$ (seven sub-cases, up to three rules for t)
8. $n = 5, v \neq \square, w_3 \in \{\sharp, \sharp\sharp\}, |w_4| > 2$ (three sub-cases, up to three rules for t)
9. $n = 5, v \neq \square, |w_3| > 2, |w_4| > 2$ (23 sub-cases, up to four rules for t)
10. $n > 5$ (The rule itself is not simulated, instead the $\xi_{w_3} \cdots \xi_{w_{n-2}}$ is compressed into one letter $\xi_{w_3 \cdots w_{n-2}}$. Thus, the cases $n > 5$ are reduced to the cases $n \leq 5$.)

3.7 An example case

In order to show how the sub-cases of the main cases above can be derived, one example is provided. This example case is number 9.16 in section 3.13.

Example 3.6. Consider $t = (u, v, \xi_{w_1 w_2} \xi_{w_3} \xi_{w_4 w_5} w) \in T_2$, so $n = 5$. Assume $|w_3| > 2$ and $|w_4| > 2$. We know $\hat{w}_2 \hat{w}_3 \hat{w}_4 = \overline{u}$. Furthermore, decompose u and v into letters: $u = a_1 \cdots a_{|u|}, v = b_1 \cdots b_{|v|}$.

Then we know there exist $i, j > 0$ such that $j > i$ and $\hat{w}_2 = \overline{a}_1 \cdots \overline{a}_i$, $\hat{w}_3 = \overline{a}_{i+1} \cdots \overline{a}_j$, and $\hat{w}_4 = \overline{a}_{j+1} \cdots \overline{a}_{|u|}$.

We use i and j to determine how letters have to be spread over the subscripts of the new compression letters. First, we compute indices k and l which would simply assign the letters of v from right to left and from the third to the first compression letter. Let $k := |v| - |u| + i$ and $l := |v| - |u| + j$. Obviously, $l > k$ always holds. In figure 3.3 the relation

between i, j, k, and l is illustrated. As one can see, l and k allow to determine over how many compression letters the reduction result can be spread with respect to w_2, w_3, and w_4.

$\hat{w}_2 = \overline{a}_1 \cdots \overline{a}_i$	$\hat{w}_3 = \overline{a}_{i+1} \cdots \overline{a}_j$	$\hat{w}_4 = \overline{a}_{j+1} \cdots \overline{a}_{\lvert u \rvert}$	
$\overline{b}_1 \cdots \overline{b}_i$	$\overline{b}_{i+1} \cdots \overline{b}_j$	$\overline{b}_{j+1} \cdots \overline{b}_{\lvert v \rvert}$	$l > k > 0$ (3 comp. l.)
$\overline{b}_1 \cdots \overline{b}_i$	$\overline{b}_{i+1} \cdots \overline{b}_{\lvert v \rvert}$		$l > 0, k \leq 0$ (2 compression letters)
$\overline{b}_1 \cdots \overline{b}_{\lvert v \rvert}$			$l, k \leq 0$ (1 compression letter)

Fig. 3.3: Identifying sub-cases for rule simulation with i, j, k, and l

Additionally, we have to consider the $\sharp$'s at the split points between w_2 and w_3, respectively between w_3 and w_4. It is clear that there exist w_2', w_2'', w_3', w_3'', w_3''', w_4', and w_4'' such that $w_2 = w_2' w_2''$, $w_3 = w_3' w_3'' w_3'''$, $w_4 = w_4' w_4''$, with $w_2'' w_3' = \sharp\sharp$ and $w_3''' w_4' = \sharp\sharp$.

Now the sub-cases depend on $k > 0$ or not, $l > 0$ or not, the length of w_2'', and the length of w_3'''. Example 3.5 above leads to $k = 1$, $l = 2$, $w_2'' = \square$, and $w_3''' = \sharp\sharp$. Generally, in this case we introduce a new nonterminal ξ_t (for each t a separate one) which is added to Γ_2 and the following four rules:

$$
\begin{aligned}
r_{t,1} &:= (\xi_{w_1 w_2} \xi_{w_3} \xi_{w_4 w_5} w, \xi_{w_1 w_2} \xi_{w_3} \xi_t w) \\
r_{t,2} &:= (\xi_{w_1 w_2} \xi_{w_3} \xi_t w, \xi_{w_1 \overline{b}_1 (\sharp\sharp b_2) \cdots (\sharp\sharp b_k) \sharp\sharp} \xi_{w_3} \xi_t w) \\
r_{t,3} &:= (\xi_{w_1 \overline{b}_1 (\sharp\sharp b_2) \cdots (\sharp\sharp b_k) \sharp\sharp} \xi_{w_3} \xi_t w, \\
&\qquad \xi_{w_1 \overline{b}_1 (\sharp\sharp b_2) \cdots (\sharp\sharp b_k) \sharp\sharp} \xi_{b_{k+1} (\sharp\sharp b_{k+2}) \cdots (\sharp\sharp b_{l+1})} \xi_t w) \\
r_{t,4} &:= (\xi_{w_1 \overline{b}_1 (\sharp\sharp b_2) \cdots (\sharp\sharp b_k) \sharp\sharp} \xi_{b_{k+1} (\sharp\sharp b_{k+2}) \cdots (\sharp\sharp b_{l+1})} \xi_t w, \\
&\qquad \xi_{w_1 \overline{b}_1 (\sharp\sharp b_2) \cdots (\sharp\sharp b_k) \sharp\sharp} \xi_{b_{k+1} (\sharp\sharp b_{k+2}) \cdots (\sharp\sharp b_{l+1})} \xi_{\sharp\sharp (b_{l+2} \sharp\sharp) \cdots (b_{\lvert v \rvert} \sharp\sharp) w_5} w).
\end{aligned}
$$

Figure 3.4 illustrates the distribution of letters in the simulation, the length of the boxes is indicating their weight. Of the surplus letters $\sharp$ only the most important are explicitly shown.

Observe that as soon as the ξ_t is introduced, the distance between the last compression letter containing a letter from $\overline{\Gamma}$ and the ξ_t is less than two. Therefore, until the ξ_t is removed again, no further simulation of a rule can be started.

$w_2 = \overline{a}_1\sharp\sharp\cdots\overline{a}_i$	$w_3 = \sharp\sharp\overline{a}_{i+1}\cdots\overline{a}_j\sharp\sharp$	$w_4 = \overline{a}_{j+1}\cdots\overline{a}_{\lvert u\rvert}\sharp\sharp$	left-hand side of $r_{t,1}$
$w_2 = \overline{a}_1\sharp\sharp\cdots\overline{a}_i$	$w_3 = \sharp\sharp\overline{a}_{i+1}\cdots\overline{a}_j\sharp\sharp$	ξ_t	right-hand side of $r_{t,1}$
$\overline{b}_1\sharp\sharp\cdots\overline{b}_k\sharp\sharp$	$w_3 = \sharp\sharp\overline{a}_{i+1}\cdots\overline{a}_j\sharp\sharp$	ξ_t	right-hand side of $r_{t,2}$
$\overline{b}_1\sharp\sharp\cdots\overline{b}_k\sharp\sharp$	$\overline{b}_{k+1}\sharp\sharp\cdots\overline{b}_{l+1}$	ξ_t	right-hand side of $r_{t,3}$
$\overline{b}_1\sharp\sharp\cdots\overline{b}_k\sharp\sharp$	$\overline{b}_{k+1}\sharp\sharp\cdots\overline{b}_{l+1}$	$\sharp\sharp\overline{b}_{l+2}\cdots\overline{b}_{\lvert v\rvert}\sharp\sharp$	right-hand side of $r_{t,4}$

Fig. 3.4: Distribution of letters in the simulation

3.8 Confluence of the simulation

Confluence of R_4 is ensured by three properties:

1. The place of any possible next reduction is uniquely given by the last over-lined index letter.
2. We assumed without restriction of generality that the original rewriting system R is normalized. Therefore, for all $t, t' \in T_1 \cup T_2, t = (u, v, w), t' = (u', v', w')$ we know $w = w' \Longrightarrow u = u' \wedge v = v' \Longrightarrow t = t'$.
   This implies that whenever the over-lined index letters form a reducible word, exactly one t can be used.
3. After introducing the ξ_t, the only possible reduction steps are given by the simulation of the original rule. This is ensured by the lookahead: ξ_t cannot be further right from the compression letter contain-

ing the last over-lined letter than two compression letters. Therefore as soon as ξ_t is introduced, no other simulation can start before it is removed again. Also, no single step of the simulation can be applied twice, and the order of applying the simulation rules is fixed. This holds because the structure of the index words w.r.t. the $\sharp$'s prevents permutations of the rules.

So, R_4 itself is confluent (there can be no critical pairs, compare to the proof of theorem 2.9).

Remark 3.6. As we will see later, assuming a normalized rewriting system is not only important for confluence but also for equivalence of the newly constructed csCRLS to the original CRLS.

3.9 Weight reduction

In order to show that R_4 is weight reducing, we need a suitable weight function. The idea is to distribute the weights of the right-hand rule sides v in the original system R over two or more compression letters. Therefore, the strategy for the construction is called "weight-spreading". The most important part of a weight function ψ are the weights defined for letters from Γ_1. The weights for Σ, $\{¢, \$\}$, and for Γ_2 can be easily found based on this. Let $\varphi(x) : W_\sharp \to \mathbb{N}$ be the weight function defined by:

$$\varphi(x) := \begin{cases} 2\mu_l + 2 & (x \in \Gamma) \\ 2\mu_l & \text{else} \end{cases}$$

Then $\psi(x) : \Gamma_1 \to \mathbb{N}$ is defined by $\psi(\xi_w) := \varphi(w) + 1$. The following property can be verified easily:

Claim. For all $\xi_v, \xi_w \in \Gamma_1$: $|v| > |w| \implies \psi(\xi_v) > \psi(\xi_w)$ and $\xi_{vw} \in \Gamma_1 \implies \psi(\xi_v) + \psi(\xi_w) > \psi(\xi_{vw})$.

So, ψ is the Γ_1 part of the required weight function. For Γ_2 the fact can be used that all Γ_1 weights are odd, so blocking letters from Γ_2 will have even weight just fitting "in between".

By giving ¢, \$, and Y the weight 1 and by assigning the weight $1 + \max\{\psi(x) | x \in \Gamma_1\}$ to the terminals (Σ) we get a weight function ψ which is correct for all our rules.

3.10 Final rules

The last step is to define rules that accept the result of a reduction:

$$R_5 := \{(¢w\$, Y) | w \in \Gamma_1^{\leq 3}, \check{w} = \overline{Y}\sharp\sharp\}.$$

3.11 Correctness of the construction

Lemma 3.1. *Let* $\Gamma' := \Sigma \cup \{¢, \$, Y\} \cup \Gamma_1 \cup \Gamma_2$, $\Sigma' := \Sigma$, $R' := R_1 \cup R_2 \cup R_3 \cup R_4 \cup R_5$. *Then* $C' := (\Gamma', \Sigma', R', ¢, \$, Y)$ *is a csCRLS and* $L_{C'} = L_C$.

Proof. There are five steps necessary: Is C' well defined? Is it weight reducing? Is it confluent? Does it define the correct language? Is it context-splittable? Again, we refer the reader to the last section of this chapter for the overview of all cases.

1. To check if C' is well defined we only have to assert that nowhere a $\xi_\square$ would be necessary in R_4. By observing the relations between the subscript word lengths and n, i, j, k, and l of the sub-cases mentioned above this can be shown to be true.
2. Checking the weight reduction of the rules is a rather tedious effort, but straight forward.
3. First, observe that the first three parts together (that is, $R_1 \cup R_2 \cup R_3$) are confluent. Secondly, reduction rules of R_4 cannot overlap with those rules or with rules from R_5 (the latter because $Y \in \textsc{Irr}(R)$). Overlaps between rules within R_4 cannot happen. (Note that as soon as a ξ_t is introduced there is always exactly one next rule that can be applied until the ξ_t is removed again. Especially, no such rule with ξ_t can be applied twice.) So, finally R' is confluent.
4. All our rules in R_4 simulate the original system R. Therefore, our new system R' cannot accept words which are not in L_C. Because R is assumed to be normalized, the exact behaviour of a automaton

accepting L_C is simulated. Therefore, for each accepting computation of such an automaton, a reduction exists which simulates it. Therefore, every word in L_C is in $L_{C'}$, too, and $L_C = L_{C'}$ holds.

5. Checking the context-splittability is the easiest part, it can be verified by simply looking at all rules.

With this lemma, the proof of the normal form theorem is complete. □

Corollary 3.1. *For each CRL L there is a weight-increasing context-sensitive grammar G such that $L(G) = L$ and each rule is context-sensitive in the sense of [Cho59].*

Proof. We do not elaborate the proof in full detail, since in [NW01a] a direct grammar construction for the whole class of GCSL, which includes CRL (theorem 2.15), is given.

Let $C = (\Gamma, \Sigma, R, t_l, t_r, Y)$ be a csCRLS defining L. Basically, one has to swap the rules sides of the rewriting rules of C. There are only two technical problems.

1. Instead of the end-markers one has to mark the left-hand and right-hand ends of the sentential form and adapt all rules using those end-markers accordingly. Of course, this will lead to some more non-terminals.
2. After swapping the sides, there might be rules of the form $\Box \to v$. Because any sentential form produced by the grammar contains at least one letter, we simply use all rules of the form $x \to xv$ and $y \to vy$ where x, y are letters from the alphabets of G (but x is not a letter marking the right-hand side of the sentential form and y not a letter marking the left-hand side).

3.12 Consequences of the result

One consequence is that the information flow during reductions is underlying stronger restrictions in a csCRLS. Any movement of a letter in either direction needs at least as many rule applications as the distance to be accomplished. Although this is only a refinement of the linear time bound for the reductions in CRLSs it might be handy for proofs.

One can see that the piecewise simulation of rules leads to a rewriting system that has reducible right-hand sides, in contrast to the normalized rewriting systems.

The context-splittable normal form and the property of being normalized seem to be dual to each other: We conjecture that there is a CRL that does not have a normalized csCRLS. For example, consider a rule $(abbb, bba)$. The author knows of no way to simulate such a rule by *a single context-splittable* rule. Besides, a length reducing simulation in more than one step with a normalized systems seems to incorporate conflicting goals.[1]

Additionally, the construction of a csCRLS heavily uses weight reduction. This makes the existence of a length reducing context-splittable normal form doubtful. Anyhow, we do not conjecture that there is a CRL that has no length reducing csCRLS nor the opposite. Already our characterization result is against intuition, so there might be an even more complicated construction which shows that a length reducing csCRLS exists for each CRL.

Some more interesting questions arise because CRL and DGCSL are the same language class, characterized by sDTPDAs. Therefore, the following results can be obtained. All details are described in [NW01a].

By dropping the condition of confluence we obtain the class of the *growing context-sensitive languages* (GCSL, defined in [DW86]) from the class CRL. A normal form corresponding to the one established here for CRL also holds for GCSL (thus justifying the use of the term *context-sensitive*). This implies that the class of the *acyclic context-sensitive languages* (ACSL) coincides with GCSL (see [Bun96]).

ACSL are those languages which can be described by a context-sensitive grammar whose context-free kernel (the context-free grammar gained by stripping the context from the context-sensitive rules) is acyclic.

Theorem 3.2. *[NW01a] For each growing context-sensitive grammar G with language L_G there exists an acyclic context-sensitive grammar G' with language $L_{G'} = L_G$.*

[1] Originally, Friedrich Otto raised the question of normalized csCRLSs during Theorietag 2001 of the German Gesellschaft für Informatik.

The definition of TPDAs allows a program $\delta : (Q \times \Gamma \times \Gamma) \to (Q \times \Gamma^* \times \Gamma^*)$ (Q are the states, Γ is the work alphabet). By a construction similar to the one for GCSL it should be possible to restrict the possible rule types—even for shrinking or bounded (D)TDPAs—to the following forms:

1. $(q, A, B) \to (q', A, C)$,
2. $(q, A, B) \to (q', AC, \square)$,
3. $(q, A, B) \to (q', C, B)$,
4. $(q, A, B) \to (q', \square, CB)$,
5. $(q, A, B) \to (q', C, \square)$,
6. $(q, A, B) \to (q', \square, C)$, or
7. $(q, A, B) \to (q', \square, \square)$,

with $q, q' \in Q, A, B, C \in \Gamma$ and without limiting the expressive powers of the respective automata classes (s(D)TPDA,b(D)TPDA). This is a line of further research.

Buntrock discusses *derivation graphs* for words of a growing context-sensitive grammar [Bun96]. The same can be done for CRLSs. For csCRLSs, one can also introduce the notion of derivation trees (under some restrictions), similar to those of context free grammars (for example, see [Har78]). Since we are not going to use this we only give some examples.

Example 3.7. Let $C = (\{¢, \$, Y, a, b, c\}, \{a, b, c\}, R, ¢, \$, Y)$ be a CRLS with a rewriting system

$$R = \{(abbba, acca), (¢acca\$, Y)\}.$$

Obviously, $L_C = \{abbba, acca\}$.

For the word *abbba* a derivation graph is:

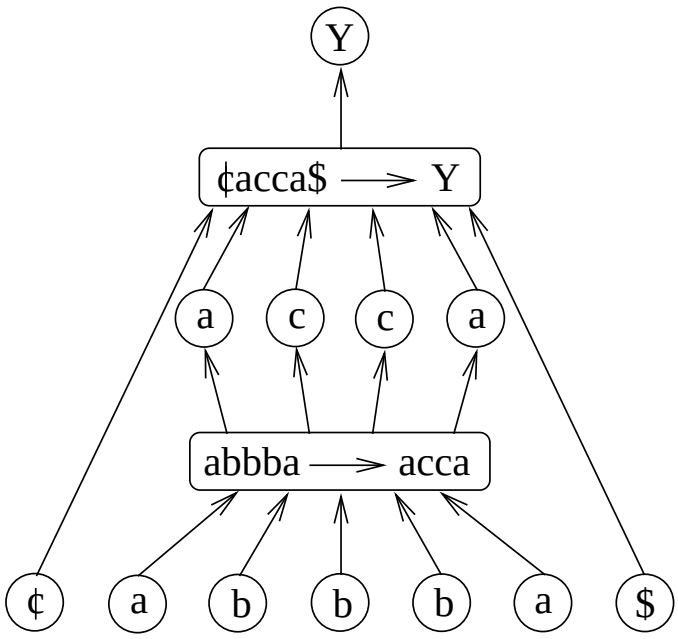

Fig. 3.5: An example for a derivation graph

Note that we have Y at the top of the graph in order to make it similar to the derivation graphs for grammars. Nevertheless, the arrows point into the direction of the rewriting rules.

As one can see, the unchanged context of a rewriting step is copied in this graph.

Now we examine a csCRLS C', which differs from C only in the rewriting rules, which are:

$$R = \{(abbba, aca), (¢aca\$, Y)\}.$$

The derivation graph for the word *abbb* still is not a tree:

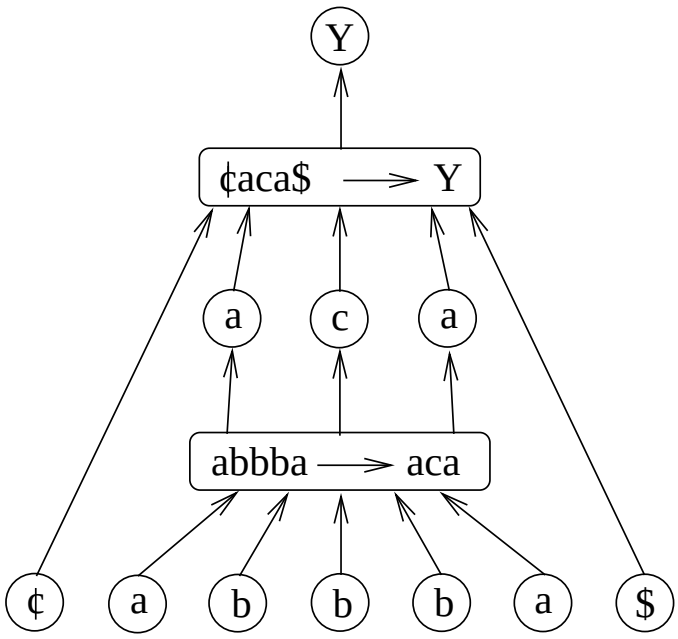

Fig. 3.6: A derivation graph of a word accepted by a csCRLS

We can remove the copied contexts, and get the following tree:

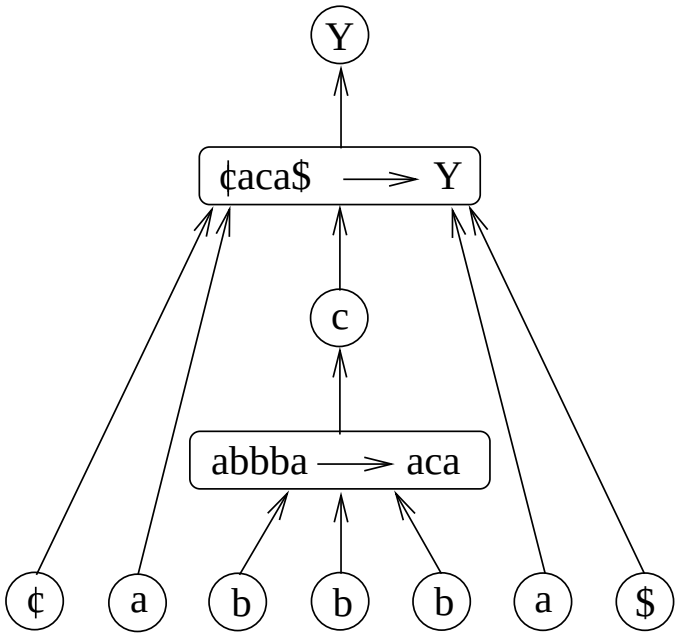

Fig. 3.7: An enriched derivation tree

This can be understood as a derivation tree which is enriched by rule information. By dropping the nodes containing the rules we get a derivation tree which only differs from a tree for a context-free grammar in the direction of the arrows:

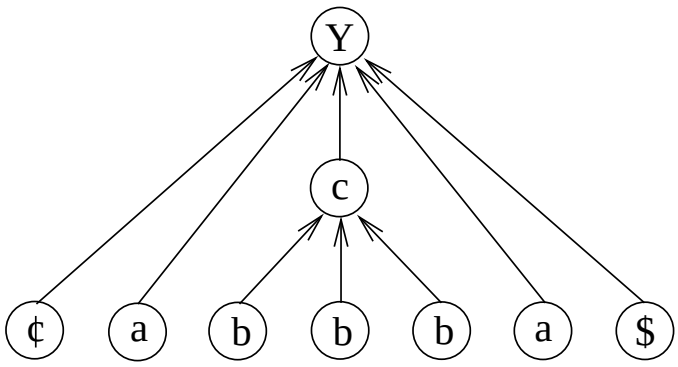

Fig. 3.8: An example for a derivation tree

It is also possible to remove the nodes ¢ and \$: Except for the last step they always are part of the context.

It is important that we cannot do these steps of removing the contexts if we have deleting rules like $(ccc, \square)$ or $(¢ \cdot c \cdot \$, ¢ \cdot \square \cdot \$)$ (the latter is a deleting rule because in context-splittings ¢ and \$ always have to be part of the contexts).

3.13 All cases for the construction of R_4

For each $t \in T_2$ we will give a set of rules that has to be added to R_4. This also uses some further nonterminals. These will be collected in the set Γ_2. Again, we assume $u = a_1 \cdots a_{|u|}, a_i \in \Gamma$ $(1 \leq i \leq |u|)$ and $v = b_1 b_2 \cdots b_{|v|}, b_i \in \Gamma$ $(1 \leq i \leq |v|)$. Note that $w_2 \neq \square$ is required by the definition of T_2.

There are ten cases which have to be handled differently:

1. The case $n = 4$ and $w_3 = \square$ is already captured by T_1: Assume $t \in T_2$ and $t = (u, v, \xi_{w_1 w_2} \xi_{w_3 w_4} w) = (u, v, \xi_{w_1 w_2} \xi_{w_4} w)$. Then there is a $w' \in \mathrm{Suff}(\xi_{w_4} w)$ such that $t' = (u, v, \xi_{w_1 w_2} w') \in T_1$. Therefore, we do not need to add rules for this case.
2. For $n = 4$, $v = \square$, $w_3 \neq \square$ we get subcases:

2.1. $w_1 = \square, w_4 = \square$, add one rule which simply deletes two letters:

$$r_t := (\xi_{w_1 w_2} \xi_{w_3 w_4} w, w).$$

2.2. $w_1 \neq \square, w_4 = \square$ uses a new nonterminal ξ_t with

$$\psi(\xi_t) = \psi(\xi_{w_3 w_4}) - 1$$

and three rules:

$$r_{t,1} := (\xi_{w_1 w_2} \xi_{w_3 w_4} w, \xi_{w_1 w_2} \xi_t w)$$
$$r_{t,2} := (\xi_{w_1 w_2} \xi_t w, \xi_{w_1} \xi_t w)$$
$$r_{t,3} := (\xi_{w_1} \xi_t w, \xi_{w_1} w).$$

2.3. $w_1 = \square, w_4 \neq \square$ uses only one rule:

$$r_t := (\xi_{w_1 w_2} \xi_{w_3 w_4} w, \xi_{w_4} w).$$

2.4. $w_1 \neq \square, w_4 \neq \square$ uses a new nonterminal ξ_t with

$$\psi(\xi_t) = \psi(\xi_{w_3 w_4}) - 1$$

and three rules:

$$r_{t,1} := (\xi_{w_1 w_2} \xi_{w_3 w_4} w, \xi_{w_1 w_2} \xi_t w)$$
$$r_{t,2} := (\xi_{w_1 w_2} \xi_t w, \xi_{w_1} \xi_t w)$$
$$r_{t,3} := (\xi_{w_1} \xi_t w, \xi_{w_1} \xi_{w_4} w).$$

3. If $n = 4$, $v \neq \square$, and $w_3 \in \{\sharp, \sharp\sharp\}$, all the action takes place in the first letter. For w_2 there exists a splitting such that $w_2 = w_2' w_2''$ with $w_2'' w_3 = \sharp\sharp$. Add the rule:

$$r_t := (\xi_{w_1 w_2} \xi_{w_3 w_4} w, \xi_{w_1 \overline{b}_1 (\sharp\sharp b_2) \cdot (\sharp\sharp b_{|v|}) w_2''} \xi_{w_3 w_4} w).$$

This case uses no new nonterminals.

4. If $n = 4$, $v \neq \square$, and $|w_3| > 2$ (then w_3 contains at least one letter from $\overline{\Gamma}$), the reduction must be split over two nonterminals. For w_2 and w_3 there exist splittings such that $w_2 = w_2' w_2''$ and $w_3 = w_3' w_3''$ with $w_2'' w_3' = \sharp\sharp$. Since both w_2 and w_3 are nonempty, we know that there exists an $i, 1 \leq i < |u|$ with $w_2' = \overline{a}_1 \sharp\sharp \cdots \overline{a}_i$ and $w_3'' = \overline{a}_{i+1} \sharp\sharp \cdots \overline{a}_{|u|} \sharp\sharp$. Now there are five sub-cases, depending on the length of w_1, w_2'', and i:
   Let $k := |v| - |u| + i$. If $k > 0$ this will be used to calculate a split point for the compressed word which is to be substituted. In some cases we use a new nonterminal ξ_t, which will be added to Γ_2. Let the weight of ξ_t be $\psi(\xi_t) := \psi(\xi_{w_3 w_4}) - 1$.

4.1. If $k \leq 0$ and $w_1 = \square$ add the rule

$$r_t := (\xi_{w_1 w_2} \xi_{w_3 w_4} w, \xi_{(\overline{b}_1 \sharp\sharp)(b_2 \sharp\sharp) \cdots (b_{|v|} \sharp\sharp) w_4} w).$$

4.2. If $k \leq 0$, $w_2'' = \square$, and $w_1 \neq \square$ add the following three rules:

$$\begin{aligned} r_{t,1} &:= (\xi_{w_1 w_2} \xi_{w_3 w_4} w, \xi_{w_1 w_2} \xi_t w) \\ r_{t,2} &:= (\xi_{w_1 w_2} \xi_t w, \xi_{w_1} \xi_t w) \\ r_{t,3} &:= (\xi_{w_1} \xi_t w, \xi_{w_1} \xi_{(\overline{b}_1 \sharp\sharp)(b_2 \sharp\sharp) \cdots (b_{|v|} \sharp\sharp) w_4} w). \end{aligned}$$

4.3. If $k > 0$ and $w_2'' = \square$ add three rules:

$$\begin{aligned} r_{t,1} &:= (\xi_{w_1 w_2} \xi_{w_3 w_4} w, \xi_{w_1 w_2} \xi_t w) \\ r_{t,2} &:= (\xi_{w_1 w_2} \xi_t w, \xi_{w_1 (\overline{b}_1 \sharp\sharp)(b_2 \sharp\sharp) \cdots (b_k \sharp\sharp)} \xi_t w) \\ r_{t,3} &:= (\xi_{w_1 (\overline{b}_1 \sharp\sharp)(b_2 \sharp\sharp) \cdots (b_k \sharp\sharp)} \xi_t w, \\ &\qquad \xi_{w_1 (\overline{b}_1 \sharp\sharp)(b_2 \sharp\sharp) \cdots (b_k \sharp\sharp)} \xi_{(b_{k+1} \sharp\sharp) \cdots (b_{|v|} \sharp\sharp) w_4} w). \end{aligned}$$

4.4. If $k \leq 0$, $w_2'' \neq \square$, and $w_1 \neq \square$ add the following three rules:

$$\begin{aligned} r_{t,1} &:= (\xi_{w_1 w_2} \xi_{w_3 w_4} w, \xi_{w_1 w_2} \xi_t w) \\ r_{t,2} &:= (\xi_{w_1 w_2} \xi_t w, \xi_{w_1 \overline{b}_1} \xi_t w) \\ r_{t,3} &:= (\xi_{w_1 \overline{b}_1} \xi_t w, \xi_{w_1} \xi_{\sharp\sharp (b_2 \sharp\sharp) \cdots (b_{|v|} \sharp\sharp) w_4} w). \end{aligned}$$

4.5. If $k > 0$ and $w_2'' \neq \square$ add:

$$r_{t,1} := (\xi_{w_1 w_2}\xi_{w_3 w_4} w, \xi_{w_1 w_2}\xi_t w)$$
$$r_{t,2} := (\xi_{w_1 w_2}\xi_t w, \xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_t w)$$
$$r_{t,3} := (\xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_t w,$$
$$\xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_{(\sharp\sharp b_{k+2})\cdots(\sharp\sharp b_{|v|})\sharp\sharp w_4} w).$$

Observe the lengths of the subscripts, they guarantee the weight reduction obtained by the rules $r_{t,2}$ and $r_{t,3}$.

5. The case $n = 5$, $v \neq \square$, and $w_4 = \square$ is already captured by the cases above so we do not build new rules in this case.
6. If $n = 5$ and $v = \square$ and $w_4 \neq \square$ we get four sub-cases:

6.1. $w_1 = \square, w_5 = \square$, add one rule which simply deletes three letters:

$$r_t := (\xi_{w_1 w_2}\xi_{w_3}\xi_{w_4 w_5} w, w).$$

6.2. $w_1 \neq \square, w_5 = \square$ uses a new nonterminal ξ_t with

$$\psi(\xi_t) = \psi(\xi_{w_4 w_5}) - 1$$

and three rules:

$$r_{t,1} := (\xi_{w_1 w_2}\xi_{w_3}\xi_{w_4 w_5} w, \xi_{w_1 w_2}\xi_{w_3}\xi_t w)$$
$$r_{t,2} := (\xi_{w_1 w_2}\xi_{w_3}\xi_t w, \xi_{w_1}\xi_{w_3}\xi_t w)$$
$$r_{t,3} := (\xi_{w_1}\xi_{w_3}\xi_t w, \xi_{w_1} w).$$

6.3. $w_1 = \square, w_5 \neq \square$ uses only one rule:

$$r_t := (\xi_{w_1 w_2}\xi_{w_3}\xi_{w_4 w_5} w, \xi_{w_5} w).$$

6.4. $w_1 \neq \square, w_5 \neq \square$ uses a new nonterminal ξ_t with

$$\psi(\xi_t) = \psi(\xi_{w_4 w_5}) - 1$$

and three rules:

$$r_{t,1} := (\xi_{w_1 w_2}\xi_{w_3}\xi_{w_4 w_5} w, \xi_{w_1 w_2}\xi_{w_3}\xi_t w)$$
$$r_{t,2} := (\xi_{w_1 w_2}\xi_{w_3}\xi_t w, \xi_{w_1}\xi_{w_3}\xi_t w)$$
$$r_{t,3} := (\xi_{w_1}\xi_{w_3}\xi_t w, \xi_{w_1}\xi_{w_5} w).$$

7. For $n = 5$, $v \neq \square$, and $w_4 \in \{\sharp, \sharp\sharp\}$ the complete reduction takes place in the first two nonterminals. This is similar to $n = 4$, $v \neq \square$, These are the sub-cases:

7.1. $w_3 = \sharp$. Then we can make a one rule reduction without new nonterminal. Add the rule:

$$(\xi_{w_1 w_2} \xi_{w_3} \xi_{w_4 w_5} w, \xi_{w_1 \overline{b}_1 (\sharp\sharp b_2) \cdots (\sharp\sharp b_{|v|}) w_3} \xi_{w_4 w_5} w).$$

7.2. $w_3 \neq \sharp$. Then w_3 contains at least one letter from $\overline{\Gamma}$, since $w_3 = \sharp\sharp$ would imply $w_4 \notin \{\sharp, \sharp\sharp\}$.
For w_2, w_3 and w_4 there exist splittings such that $w_2 = w_2' w_2''$, $w_3 = w_3' w_3'' w_3'''$ and $w_4 = w_4' w_4''$ with $w_2'' w_3' = \sharp\sharp$ and $w_3''' w_4' = \sharp\sharp$. Since both w_2 and w_3 are nonempty, we know that there exists an $i, 1 \leq i < |u|$ with $w_2' = \overline{a}_1 \sharp\sharp \cdots \overline{a}_i$ and $w_3'' = \overline{a}_{i+1} \sharp\sharp \cdots \overline{a}_{|u|}$. Now there are five sub-cases, depending on the length of w_1, w_2'', and i: Let $k := |v| - |u| + i$. If $k > 0$ this will be used to calculate a split point for the compressed word which is to be substituted. In some cases we use a new nonterminal ξ_t, which will be added to Γ_2. Let the weight of ξ_t be $\psi(\xi_t) := \psi(\xi_{w_3}) - 1$.

7.2.1. If $k \leq 0$ and $w_1 = \square$ add the rule

$$r_t := (\xi_{w_1 w_2} \xi_{w_3} \xi_{w_4 w_5} w, \xi_{\overline{b}_1 (\sharp\sharp b_2) \cdots (\sharp\sharp b_{|v|}) w_3'''} \xi_{w_4 w_5} w).$$

7.2.2. If $k \leq 0$, $w_2'' = \square$, and $w_1 \neq \square$ add the following three rules:

$$\begin{aligned}
r_{t,1} &:= (\xi_{w_1 w_2} \xi_{w_3} \xi_{w_4 w_5} w, \xi_{w_1 w_2} \xi_t \xi_{w_4 w_5} w) \\
r_{t,2} &:= (\xi_{w_1 w_2} \xi_t \xi_{w_4 w_5} w, \xi_{w_1} \xi_t \xi_{w_4 w_5} w) \\
r_{t,3} &:= (\xi_{w_1} \xi_t \xi_{w_4 w_5} w, \xi_{w_1} \xi_{\overline{b}_1 (\sharp\sharp b_2) \cdots (\sharp\sharp b_{|v|}) w_3'''} \xi_{w_4 w_5} w).
\end{aligned}$$

7.2.3. If $k > 0$ and $w_2'' = \square$ add three rules:

$$\begin{aligned}
r_{t,1} &:= (\xi_{w_1 w_2} \xi_{w_3} \xi_{w_4 w_5} w, \xi_{w_1 w_2} \xi_t \xi_{w_4 w_5} w) \\
r_{t,2} &:= (\xi_{w_1 w_2} \xi_t \xi_{w_4 w_5} w, \xi_{w_1 (\overline{b}_1 \sharp\sharp)(b_2 \sharp\sharp) \cdots (b_k \sharp\sharp)} \xi_t \xi_{w_4 w_5} w) \\
r_{t,3} &:= (\xi_{w_1 (\overline{b}_1 \sharp\sharp)(b_2 \sharp\sharp) \cdots (b_k \sharp\sharp)} \xi_t \xi_{w_4 w_5} w, \\
&\qquad \xi_{w_1 (\overline{b}_1 \sharp\sharp)(b_2 \sharp\sharp) \cdots (b_k \sharp\sharp)} \xi_{b_{k+1} (\sharp\sharp b_{k+2}) \cdots (\sharp\sharp b_{|v|}) w_3'''} \xi_{w_4 w_5} w).
\end{aligned}$$

7.2.4. If $k \leq 0$, $w_2'' \neq \square$, and $w_1 \neq \square$ add the following three rules:

$$r_{t,1} := (\xi_{w_1w_2}\xi_{w_3}\xi_{w_4w_5}w, \xi_{w_1w_2}\xi_t\xi_{w_4w_5}w)$$
$$r_{t,2} := (\xi_{w_1w_2}\xi_t\xi_{w_4w_5}w, \xi_{w_1\overline{b}_1}\xi_t\xi_{w_4w_5}w)$$
$$r_{t,3} := (\xi_{w_1\overline{b}_1}\xi_t\xi_{w_4w_5}w, \xi_{w_1\overline{b}_1}\xi_{(\sharp\sharp b_2)\cdots(\sharp\sharp b_{|v|})}w_3'''\xi_{w_4w_5}w).$$

7.2.5. If $k > 0$ and $w_2'' \neq \square$ add:

$$r_{t,1} := (\xi_{w_1w_2}\xi_{w_3}\xi_{w_4w_5}w, \xi_{w_1w_2}\xi_t\xi_{w_4w_5}w)$$
$$r_{t,2} := (\xi_{w_1w_2}\xi_t\xi_{w_4w_5}w, \xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_t\xi_{w_4w_5}w)$$
$$r_{t,3} := (\xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_t\xi_{w_4w_5}w,$$
$$\xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_{(\sharp\sharp b_{k+2})\cdots(\sharp\sharp b_{|v|})}w_3'''\xi_{w_4w_5}w).$$

8. For $n = 5$, $v \neq \square$, $w_3 \in \{\sharp, \sharp\sharp\}$, and $w_4 \notin \{\square, \sharp, \sharp\sharp\}$ there are less sub-cases. Then w_4 contains at least one letter from $\overline{\Gamma}$, since $|w_4| > 2$. For w_2, w_3 and w_4 there exist splittings such that $w_2 = w_2'w_2''$, $w_3 = w_3'w_3''w_3'''$ and $w_4 = w_4'w_4''$ with $w_2''w_3' = \sharp\sharp$ and $w_3'''w_4' = \sharp\sharp$. Since both w_2 and w_4 are nonempty, we know that there exists an $i, 1 \leq i < |u|$ with $w_2' = \overline{a}_1\sharp\sharp\cdots\overline{a}_i$ and $w_4'' = \overline{a}_{i+1}\sharp\sharp\cdots\overline{a}_{|u|}\sharp\sharp$. Now there are three sub-cases, depending on the length of w_1, w_2'', and i: Let $k := |v| - |u| + i$. If $k > 0$ this will be used to calculate a split point for the compressed word which is to be substituted. In some cases we use a new nonterminal ξ_t, which will be added to Γ_2. Let the weight of ξ_t be $\psi(\xi_t) := \psi(\xi_{w_4w_5}) - 1$.

8.1. If $k \leq 0$ and $w_1 = \square$ add the rule

$$r_t := (\xi_{w_1w_2}\xi_{w_3}\xi_{w_4w_5}w, \xi_{(\overline{b}_1\sharp\sharp)(b_2\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5}w).$$

8.2. If $k \leq 0$ and $w_1 \neq \square$ add the following three rules:

$$r_{t,1} := (\xi_{w_1w_2}\xi_{w_3}\xi_{w_4w_5}w, \xi_{w_1w_2}\xi_{w_3}\xi_t w)$$
$$r_{t,2} := (\xi_{w_1w_2}\xi_{w_3}\xi_t w, \xi_{w_1\overline{b}_1}\xi_t w)$$
$$r_{t,3} := (\xi_{w_1\overline{b}_1}\xi_t w, \xi_{w_1\overline{b}_1}\xi_{\sharp\sharp(b_2\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5}w).$$

8.3. If $k > 0$ add:

$$r_{t,1} := (\xi_{w_1w_2}\xi_{w_3}\xi_{w_4w_5}w, \xi_{w_1w_2}\xi_{w_3}\xi_t w)$$
$$r_{t,2} := (\xi_{w_1w_2}\xi_{w_3}\xi_t w, \xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_t w)$$
$$r_{t,3} := (\xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_t w,$$
$$\xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_{(\sharp\sharp b_{k+2})\cdots(\sharp\sharp b_{|v|})\sharp\sharp w_5}w).$$

9. For $n = 5$, $v \neq \Box$, $w_3 \notin \{\sharp, \sharp\sharp\}$ (w_3 cannot be empty), and $w_4 \notin \{\Box, \sharp, \sharp\sharp\}$ we will have the biggest collection of sub-cases. In this case w_2,w_3, and w_4 contain at least one letter from $\overline{\Gamma}$.
For w_2, w_3 and w_4 there exist splittings such that $w_2 = w_2'w_2''$, $w_3 = w_3'w_3''w_3'''$ and $w_4 = w_4'w_4''$ with $w_2''w_3' = \sharp\sharp$ and $w_3'''w_4' = \sharp\sharp$. Since both w_2, w_3 and w_4 are nonempty, we know that there exist an $i, j, 1 \leq i < j < |u|$ with $w_2' = \overline{a}_1\sharp\sharp\cdots\overline{a}_i$, $w_3'' = \overline{a}_{i+1}\sharp\sharp\cdots\overline{a}_j$, and $w_4'' = \overline{a}_{j+1}\sharp\sharp\cdots\overline{a}_{|u|}\sharp\sharp$. Now there are 23 sub-cases(!), depending on the length of w_1, w_2'', w_3''', i, and j:
Let $k := |v| - |u| + i$. If $k > 0$ this will be used to calculate a split point for the compressed word which is to be substituted. Similarly, we will use $l := |v| - |u| + j$. Note that $l > 0$ implies $|v| \geq 2$.
In some cases we use a new nonterminal ξ_t, which will be added to Γ_2. Let the weight of ξ_t be $\psi(\xi_t) := \psi(\xi_{w_4w_5}) - 1$.

9.1. $k \leq 0$, $l \leq 0$, and $w_1 = \Box$. We only use a single rule:

$$(\xi_{w_1w_2}\xi_{w_3}\xi_{w_4w_5}w, \xi_{(\overline{b_1}\sharp\sharp)(b_2\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5}w).$$

9.2. $k \leq 0$, $l > 0$, $w_1 = \Box$, and $w_3''' = \Box$. We add three rules:

$$r_{t,1} := (\xi_{w_1w_2}\xi_{w_3}\xi_{w_4w_5}w, \xi_{w_1w_2}\xi_{w_3}\xi_t w)$$
$$r_{t,2} := (\xi_{w_1w_2}\xi_{w_3}\xi_t w, \xi_{\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_l)\sharp}\xi_t w)$$
$$r_{t,3} := (\xi_{\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_l)\sharp}\xi_t w, \xi_{\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_l)\sharp}\xi_{\sharp(b_{l+1}\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5}w).$$

9.3. $k \leq 0$, $l > 0$, $w_1 = \Box$, and $w_3''' \neq \Box$. Again, add three rules:

$$r_{t,1} := (\xi_{w_1w_2}\xi_{w_3}\xi_{w_4w_5}w, \xi_{w_1w_2}\xi_{w_3}\xi_t w)$$
$$r_{t,2} := (\xi_{w_1w_2}\xi_{w_3}\xi_t w, \xi_{\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{l+1})}\xi_t w)$$
$$r_{t,3} := (\xi_{\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{l+1})}\xi_t w, \xi_{\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{l+1})}\xi_{(\sharp\sharp b_{l+2})\cdots(\sharp\sharp b_{|v|})\sharp\sharp w_5}w).$$

9.4. $k \leq 0$, $l \leq 0$, and $w_1 \neq \Box$. We will have three new rules:

$$
\begin{aligned}
r_{t,1} &:= (\xi_{w_1w_2}\xi_{w_3}\xi_{w_4w_5}w, \xi_{w_1w_2}\xi_{w_3}\xi_t w)\\
r_{t,2} &:= (\xi_{w_1w_2}\xi_{w_3}\xi_t w, \xi_{w_1}\xi_{w_3}\xi_t w)\\
r_{t,3} &:= (\xi_{w_1}\xi_{w_3}\xi_t w, \xi_{w_1}\xi_{(\overline{b}_1\sharp\sharp)(b_2\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5}w).
\end{aligned}
$$

9.5. $k \leq 0$, $l > 0$, $w_1 \neq \Box$, $w_2'' = \Box$, and $w_3''' = \Box$. Add four rules:

$$
\begin{aligned}
r_{t,1} &:= (\xi_{w_1w_2}\xi_{w_3}\xi_{w_4w_5}w, \xi_{w_1w_2}\xi_{w_3}\xi_t w)\\
r_{t,2} &:= (\xi_{w_1w_2}\xi_{w_3}\xi_t w, \xi_{w_1}\xi_{w_3}\xi_t w)\\
r_{t,3} &:= (\xi_{w_1}\xi_{w_3}\xi_t w, \xi_{w_1}\xi_{\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{j-i})\sharp}\xi_t w)\\
r_{t,4} &:= (\xi_{w_1}\xi_{\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{j-i})\sharp}\xi_t w,\\
&\qquad \xi_{w_1}\xi_{\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{j-i})\sharp}\xi_{\sharp(b_{j-i+1}\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5}w).
\end{aligned}
$$

9.6. $k \leq 0$, $l > 0$, $w_1 \neq \Box$, $w_2'' = \sharp$, and $w_3''' = \Box$. Add four rules:

$$
\begin{aligned}
r_{t,1} &:= (\xi_{w_1w_2}\xi_{w_3}\xi_{w_4w_5}w, \xi_{w_1w_2}\xi_{w_3}\xi_t w)\\
r_{t,2} &:= (\xi_{w_1w_2}\xi_{w_3}\xi_t w, \xi_{w_1\overline{b}_1}\xi_{w_3}\xi_t w)\\
r_{t,3} &:= (\xi_{w_1\overline{b}_1}\xi_{w_3}\xi_t w, \xi_{w_1\overline{b}_1}\xi_{\sharp(\sharp b_2\sharp)\cdots(\sharp b_l\sharp)}\xi_t w)\\
r_{t,4} &:= (\xi_{w_1\overline{b}_1}\xi_{\sharp(\sharp b_2\sharp)\cdots(\sharp b_l\sharp)}\xi_t w,\\
&\qquad \xi_{w_1\overline{b}_1}\xi_{\sharp(\sharp b_2\sharp)\cdots(\sharp b_l\sharp)}\xi_{\sharp(b_{l+1}\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5}w).
\end{aligned}
$$

9.7. $k \leq 0$, $l > 0$, $w_1 \neq \Box$, $w_2'' = \sharp\sharp$, and $w_3''' = \Box$. Add four rules:

$$
\begin{aligned}
r_{t,1} &:= (\xi_{w_1w_2}\xi_{w_3}\xi_{w_4w_5}w, \xi_{w_1w_2}\xi_{w_3}\xi_t w)\\
r_{t,2} &:= (\xi_{w_1w_2}\xi_{w_3}\xi_t w, \xi_{w_1\overline{b}_1\sharp}\xi_{w_3}\xi_t w)\\
r_{t,3} &:= (\xi_{w_1\overline{b}_1\sharp}\xi_{w_3}\xi_t w, \xi_{w_1\overline{b}_1\sharp}\xi_{(\sharp b_2\sharp)\cdots(\sharp b_l\sharp)}\xi_t w)\\
r_{t,4} &:= (\xi_{w_1\overline{b}_1\sharp}\xi_{(\sharp b_2\sharp)\cdots(\sharp b_l\sharp)}\xi_t w,\\
&\qquad \xi_{w_1\overline{b}_1\sharp}\xi_{(\sharp b_2\sharp)\cdots(\sharp b_l\sharp)}\xi_{\sharp(b_{l+1}\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5}w).
\end{aligned}
$$

9.8. $k \leq 0$, $l > 0$, $w_1 \neq \Box$, $w_2'' = \Box$, and $w_3''' = \sharp$. Add four rules:

$$
\begin{aligned}
r_{t,1} &:= (\xi_{w_1w_2}\xi_{w_3}\xi_{w_4w_5}w, \xi_{w_1w_2}\xi_{w_3}\xi_t w)\\
r_{t,2} &:= (\xi_{w_1w_2}\xi_{w_3}\xi_t w, \xi_{w_1}\xi_{w_3}\xi_t w)\\
r_{t,3} &:= (\xi_{w_1}\xi_{w_3}\xi_t w, \xi_{w_1}\xi_{\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{j-i})\sharp\sharp}\xi_t w)\\
r_{t,4} &:= (\xi_{w_1}\xi_{\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{j-i})\sharp\sharp}\xi_t w,\\
&\qquad \xi_{w_1}\xi_{\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{j-i})\sharp\sharp}\xi_{(b_{j-i+1}\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5}w).
\end{aligned}
$$

9.9. $k \leq 0$, $l > 0$, $w_1 \neq \Box$, $w_2'' = \sharp$, and $w_3''' = \sharp$. Add four rules:

$$
\begin{aligned}
r_{t,1} &:= (\xi_{w_1w_2}\xi_{w_3}\xi_{w_4w_5}w, \xi_{w_1w_2}\xi_{w_3}\xi_t w)\\
r_{t,2} &:= (\xi_{w_1w_2}\xi_{w_3}\xi_t w, \xi_{w_1\overline{b}_1}\xi_{w_3}\xi_t w)\\
r_{t,3} &:= (\xi_{w_1\overline{b}_1}\xi_{w_3}\xi_t w, \xi_{w_1\overline{b}_1}\xi_{\sharp(\sharp b_2\sharp)\cdots(\sharp b_l\sharp)\sharp}\xi_t w)\\
r_{t,4} &:= (\xi_{w_1\overline{b}_1}\xi_{\sharp(\sharp b_2\sharp)\cdots(\sharp b_l\sharp)\sharp}\xi_t w,\\
&\qquad \xi_{w_1\overline{b}_1}\xi_{\sharp(\sharp b_2\sharp)\cdots(\sharp b_l\sharp)\sharp}\xi_{(b_{l+1}\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5}w).
\end{aligned}
$$

9.10. $k \leq 0$, $l > 0$, $w_1 \neq \Box$, $w_2'' = \sharp\sharp$, and $w_3''' = \sharp$. Add four rules:

$$
\begin{aligned}
r_{t,1} &:= (\xi_{w_1w_2}\xi_{w_3}\xi_{w_4w_5}w, \xi_{w_1w_2}\xi_{w_3}\xi_t w)\\
r_{t,2} &:= (\xi_{w_1w_2}\xi_{w_3}\xi_t w, \xi_{w_1\overline{b}_1\sharp}\xi_{w_3}\xi_t w)\\
r_{t,3} &:= (\xi_{w_1\overline{b}_1\sharp}\xi_{w_3}\xi_t w, \xi_{w_1\overline{b}_1\sharp}\xi_{(\sharp b_2\sharp)\cdots(\sharp b_l\sharp)\sharp}\xi_t w)\\
r_{t,4} &:= (\xi_{w_1\overline{b}_1\sharp}\xi_{(\sharp b_2\sharp)\cdots(\sharp b_l\sharp)\sharp}\xi_t w,\\
&\qquad \xi_{w_1\overline{b}_1\sharp}\xi_{(\sharp b_2\sharp)\cdots(\sharp b_l\sharp)\sharp}\xi_{(b_{l+1}\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5}w).
\end{aligned}
$$

9.11. $k \leq 0$, $l > 0$, $w_1 \neq \Box$, $w_2'' = \Box$, and $w_3''' = \sharp\sharp$. Add four rules:

$$
\begin{aligned}
r_{t,1} &:= (\xi_{w_1w_2}\xi_{w_3}\xi_{w_4w_5}w, \xi_{w_1w_2}\xi_{w_3}\xi_t w)\\
r_{t,2} &:= (\xi_{w_1w_2}\xi_{w_3}\xi_t w, \xi_{w_1}\xi_{w_3}\xi_t w)\\
r_{t,3} &:= (\xi_{w_1}\xi_{w_3}\xi_t w, \xi_{w_1}\xi_{\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{l+1})}\xi_t w)\\
r_{t,4} &:= (\xi_{w_1}\xi_{\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{l+1})}\xi_t w,\\
&\qquad \xi_{w_1}\xi_{\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{l+1})}\xi_{\sharp\sharp(b_{l+2}\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5}w).
\end{aligned}
$$

9.12. $k \leq 0$, $l > 0$, $w_1 \neq \Box$, $w_2'' = \sharp$, and $w_3''' = \sharp\sharp$. Add four rules:

$$
\begin{aligned}
r_{t,1} &:= (\xi_{w_1w_2}\xi_{w_3}\xi_{w_4w_5}w, \xi_{w_1w_2}\xi_{w_3}\xi_t w)\\
r_{t,2} &:= (\xi_{w_1w_2}\xi_{w_3}\xi_t w, \xi_{w_1\overline{b}_1}\xi_{w_3}\xi_t w)\\
r_{t,3} &:= (\xi_{w_1\overline{b}_1}\xi_{w_3}\xi_t w, \xi_{w_1\overline{b}_1}\xi_{(\sharp\sharp b_2)\cdots(\sharp\sharp b_{l+1})}\xi_t w)\\
r_{t,4} &:= (\xi_{w_1\overline{b}_1}\xi_{(\sharp\sharp b_2)\cdots(\sharp\sharp b_{l+1})}\xi_t w,\\
&\qquad \xi_{w_1\overline{b}_1}\xi_{(\sharp\sharp b_2)\cdots(\sharp\sharp b_{l+1})}\xi_{\sharp\sharp(b_{l+2}\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5}w).
\end{aligned}
$$

9.13. $k \leq 0$, $l > 0$, $w_1 \neq \Box$, $w_2'' = \sharp\sharp$, and $w_3''' = \sharp\sharp$. Add four rules:

$$
\begin{aligned}
r_{t,1} &:= (\xi_{w_1w_2}\xi_{w_3}\xi_{w_4w_5}w, \xi_{w_1w_2}\xi_{w_3}\xi_t w)\\
r_{t,2} &:= (\xi_{w_1w_2}\xi_{w_3}\xi_t w, \xi_{w_1\overline{b}_1\sharp}\xi_{w_3}\xi_t w)\\
r_{t,3} &:= (\xi_{w_1\overline{b}_1\sharp}\xi_{w_3}\xi_t w, \xi_{w_1\overline{b}_1\sharp}\xi_{(\sharp b_2\sharp)\cdots(\sharp b_{l+1}\sharp)}\xi_t w)\\
r_{t,4} &:= (\xi_{w_1\overline{b}_1\sharp}\xi_{(\sharp b_2\sharp)\cdots(\sharp b_{l+1}\sharp)}\xi_t w,\\
&\qquad \xi_{w_1\overline{b}_1\sharp}\xi_{(\sharp b_2\sharp)\cdots(\sharp b_{l+1}\sharp)}\xi_{\sharp(b_{l+2}\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5}w).
\end{aligned}
$$

9.14. $k > 0$, $l > 0$, $w_2'' = \square$, $w_3''' = \square$. Again, we use four rules:

$$\begin{aligned}
r_{t,1} &:= (\xi_{w_1 w_2}\xi_{w_3}\xi_{w_4 w_5}w, \xi_{w_1 w_2}\xi_{w_3}\xi_t w)\\
r_{t,2} &:= (\xi_{w_1 w_2}\xi_{w_3}\xi_t w, \xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_k)\sharp\sharp}\xi_{w_3}\xi_t w)\\
r_{t,3} &:= (\xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_k)\sharp\sharp}\xi_{w_3}\xi_t w,\\
&\qquad \xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_k)\sharp\sharp}\xi_{b_{k+1}(\sharp\sharp b_{k+2})\cdots(\sharp\sharp b_l)\sharp}\xi_t w)\\
r_{t,4} &:= (\xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_k)\sharp\sharp}\xi_{b_{k+1}(\sharp\sharp b_{k+2})\cdots(\sharp\sharp b_l)\sharp}\xi_t w,\\
&\qquad \xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_k)\sharp\sharp}\xi_{b_{k+1}(\sharp\sharp b_{k+2})\cdots(\sharp\sharp b_l)\sharp}\xi_{\sharp(b_{l+1}\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5}w).
\end{aligned}$$

9.15. $k > 0$, $l > 0$, $w_2'' = \square$, $w_3''' = \sharp$. Add the following rules:

$$\begin{aligned}
r_{t,1} &:= (\xi_{w_1 w_2}\xi_{w_3}\xi_{w_4 w_5}w, \xi_{w_1 w_2}\xi_{w_3}\xi_t w)\\
r_{t,2} &:= (\xi_{w_1 w_2}\xi_{w_3}\xi_t w, \xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_k)\sharp\sharp}\xi_{w_3}\xi_t w)\\
r_{t,3} &:= (\xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_k)\sharp\sharp}\xi_{w_3}\xi_t w,\\
&\qquad \xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_k)\sharp\sharp}\xi_{b_{k+1}(\sharp\sharp b_{k+2})\cdots(\sharp\sharp b_l)\sharp\sharp}\xi_t w)\\
r_{t,4} &:= (\xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_k)\sharp\sharp}\xi_{b_{k+1}(\sharp\sharp b_{k+2})\cdots(\sharp\sharp b_l)\sharp\sharp}\xi_t w,\\
&\qquad \xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_k)\sharp\sharp}\xi_{b_{k+1}(\sharp\sharp b_{k+2})\cdots(\sharp\sharp b_l)\sharp\sharp}\xi_{(b_{l+1}\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5}w).
\end{aligned}$$

9.16. $k > 0$, $l > 0$, $w_2'' = \square$, $w_3''' = \sharp\sharp$. Add the following rules:

$$\begin{aligned}
r_{t,1} &:= (\xi_{w_1 w_2}\xi_{w_3}\xi_{w_4 w_5}w, \xi_{w_1 w_2}\xi_{w_3}\xi_t w)\\
r_{t,2} &:= (\xi_{w_1 w_2}\xi_{w_3}\xi_t w, \xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_k)\sharp\sharp}\xi_{w_3}\xi_t w)\\
r_{t,3} &:= (\xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_k)\sharp\sharp}\xi_{w_3}\xi_t w,\\
&\qquad \xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_k)\sharp\sharp}\xi_{b_{k+1}(\sharp\sharp b_{k+2})\cdots(\sharp\sharp b_{l+1})}\xi_t w)\\
r_{t,4} &:= (\xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_k)\sharp\sharp}\xi_{b_{k+1}(\sharp\sharp b_{k+2})\cdots(\sharp\sharp b_{l+1})}\xi_t w,\\
&\qquad \xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_k)\sharp\sharp}\xi_{b_{k+1}(\sharp\sharp b_{k+2})\cdots(\sharp\sharp b_{l+1})}\xi_{\sharp\sharp(b_{l+2}\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5}w).
\end{aligned}$$

9.17. $k > 0$, $l > 0$, $w_2'' = \sharp$, $w_3''' = \square$. Again, we use four rules:

$$\begin{aligned}
r_{t,1} &:= (\xi_{w_1 w_2}\xi_{w_3}\xi_{w_4 w_5}w, \xi_{w_1 w_2}\xi_{w_3}\xi_t w)\\
r_{t,2} &:= (\xi_{w_1 w_2}\xi_{w_3}\xi_t w, \xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_{w_3}\xi_t w)\\
r_{t,3} &:= (\xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_{w_3}\xi_t w,\\
&\qquad \xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_{(\sharp\sharp b_{k+2})\cdots(\sharp\sharp b_l)\sharp}\xi_t w)\\
r_{t,4} &:= (\xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_{(\sharp\sharp b_{k+2})\cdots(\sharp\sharp b_l)\sharp}\xi_t w,\\
&\qquad \xi_{w_1\overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_{(\sharp\sharp b_{k+2})\cdots(\sharp\sharp b_l)\sharp}\xi_{\sharp(b_{l+1}\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5}w).
\end{aligned}$$

9.18. $k > 0$, $l > 0$, $w_2'' = \sharp$, $w_3''' = \sharp$. Again, we use four rules:

$$\begin{aligned}
r_{t,1} &:= (\xi_{w_1 w_2}\xi_{w_3}\xi_{w_4 w_5} w, \xi_{w_1 w_2}\xi_{w_3}\xi_t w)\\
r_{t,2} &:= (\xi_{w_1 w_2}\xi_{w_3}\xi_t w, \xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_{w_3}\xi_t w)\\
r_{t,3} &:= (\xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_{w_3}\xi_t w,\\
&\qquad \xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_{(\sharp\sharp b_{k+2})\cdots(\sharp\sharp b_l)\sharp\sharp}\xi_t w)\\
r_{t,4} &:= (\xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_{(\sharp\sharp b_{k+2})\cdots(\sharp\sharp b_l)\sharp\sharp}\xi_t w,\\
&\qquad \xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_{(\sharp\sharp b_{k+2})\cdots(\sharp\sharp b_l)\sharp\sharp}\xi_{(b_{l+1}\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5} w).
\end{aligned}$$

9.19. $k > 0$, $l > 0$, $w_2'' = \sharp$, $w_3''' = \sharp\sharp$. Again, we use four rules:

$$\begin{aligned}
r_{t,1} &:= (\xi_{w_1 w_2}\xi_{w_3}\xi_{w_4 w_5} w, \xi_{w_1 w_2}\xi_{w_3}\xi_t w)\\
r_{t,2} &:= (\xi_{w_1 w_2}\xi_{w_3}\xi_t w, \xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_{w_3}\xi_t w)\\
r_{t,3} &:= (\xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_{w_3}\xi_t w,\\
&\qquad \xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_{(\sharp\sharp b_{k+2})\cdots(\sharp\sharp b_{l+1})}\xi_t w)\\
r_{t,4} &:= (\xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_{(\sharp\sharp b_{k+2})\cdots(\sharp\sharp b_{l+1})}\xi_t w,\\
&\qquad \xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})}\xi_{(\sharp\sharp b_{k+2})\cdots(\sharp\sharp b_{l+1})}\xi_{\sharp\sharp(b_{l+2}\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5} w).
\end{aligned}$$

9.20. $k > 0$, $l > 0$, $w_2'' = \sharp\sharp$, $w_3''' = \Box$, $w_3 \notin \overline{\Gamma}$. Note that under these premises $l - k > 1$. Again, we use four rules:

$$\begin{aligned}
r_{t,1} &:= (\xi_{w_1 w_2}\xi_{w_3}\xi_{w_4 w_5} w, \xi_{w_1 w_2}\xi_{w_3}\xi_t w)\\
r_{t,2} &:= (\xi_{w_1 w_2}\xi_{w_3}\xi_t w, \xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})\sharp}\xi_{w_3}\xi_t w)\\
r_{t,3} &:= (\xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})\sharp}\xi_{w_3}\xi_t w,\\
&\qquad \xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})\sharp}\xi_{(\sharp b_{k+2}\sharp)\cdots(\sharp b_l\sharp)}\xi_t w)\\
r_{t,4} &:= (\xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})\sharp}\xi_{(\sharp b_{k+2}\sharp)\cdots(\sharp b_l\sharp)}\xi_t w,\\
&\qquad \xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})\sharp}\xi_{(\sharp b_{k+2}\sharp)\cdots(\sharp b_l\sharp)}\xi_{\sharp(b_{l+1}\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5} w).
\end{aligned}$$

9.21. $k > 0$, $l > 0$, $w_2'' = \sharp\sharp$, $w_3''' = \Box$, $w_3 \in \overline{\Gamma}$. Note that in this case $k + 1 = l$. This case only uses three rules:

$$\begin{aligned}
r_{t,1} &:= (\xi_{w_1 w_2}\xi_{w_3}\xi_{w_4 w_5} w, \xi_{w_1 w_2}\xi_{w_3}\xi_t w)\\
r_{t,2} &:= (\xi_{w_1 w_2}\xi_{w_3}\xi_t w, \xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})\sharp}\xi_{w_3}\xi_t w)\\
r_{t,3} &:= (\xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})\sharp}\xi_{w_3}\xi_t w,\\
&\qquad \xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})\sharp}\xi_{\sharp(b_{k+2}\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5} w).
\end{aligned}$$

9.22. $k > 0$, $l > 0$, $w_2'' = \sharp\sharp$, $w_3''' = \sharp$. Again, we use four rules:

$$\begin{aligned}
r_{t,1} &:= (\xi_{w_1 w_2}\xi_{w_3}\xi_{w_4 w_5}w, \xi_{w_1 w_2}\xi_{w_3}\xi_t w)\\
r_{t,2} &:= (\xi_{w_1 w_2}\xi_{w_3}\xi_t w, \xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})\sharp}\xi_{w_3}\xi_t w)\\
r_{t,3} &:= (\xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})\sharp}\xi_{w_3}\xi_t w,\\
&\qquad \xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})\sharp}\xi_{(\sharp b_{k+2}\sharp)\cdots(\sharp b_l\sharp)\sharp}\xi_t w)\\
r_{t,4} &:= (\xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})\sharp}\xi_{(\sharp b_{k+2}\sharp)\cdots(\sharp b_l\sharp)\sharp}\xi_t w,\\
&\qquad \xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})\sharp}\xi_{(\sharp b_{k+2}\sharp)\cdots(\sharp b_l\sharp)\sharp}\xi_{(b_{l+1}\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5}w).
\end{aligned}$$

9.23. $k > 0$, $l > 0$, $w_2'' = \sharp\sharp$, $w_3''' = \sharp\sharp$. Again, we use four rules:

$$\begin{aligned}
r_{t,1} &:= (\xi_{w_1 w_2}\xi_{w_3}\xi_{w_4 w_5}w, \xi_{w_1 w_2}\xi_{w_3}\xi_t w)\\
r_{t,2} &:= (\xi_{w_1 w_2}\xi_{w_3}\xi_t w, \xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})\sharp}\xi_{w_3}\xi_t w)\\
r_{t,3} &:= (\xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})\sharp}\xi_{w_3}\xi_t w,\\
&\qquad \xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})\sharp}\xi_{(\sharp b_{k+2}\sharp)\cdots(\sharp b_l\sharp)\sharp b_{l+1}}\xi_t w)\\
r_{t,4} &:= (\xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})\sharp}\xi_{(\sharp b_{k+2}\sharp)\cdots(\sharp b_l\sharp)\sharp b_{l+1}}\xi_t w,\\
&\qquad \xi_{w_1 \overline{b}_1(\sharp\sharp b_2)\cdots(\sharp\sharp b_{k+1})\sharp}\xi_{(\sharp b_{k+2}\sharp)\cdots(\sharp b_l\sharp)\sharp b_{l+1}}\xi_{\sharp\sharp(b_{l+2}\sharp\sharp)\cdots(b_{|v|}\sharp\sharp)w_5}w).
\end{aligned}$$

10. If $n > 5$ we only compress the information. In consequence, any reduction will take place by the rules of the cases for $n \leq 5$. Consider all $t = (u, v, \xi_{w_1 w_2}\xi_{w_3}\xi_{w_4}\cdots\xi_{w_{n-2}}\xi_{w_{n-1}w_n}w) \in T_2$ with $n \geq 6$. Then $\xi_{w_{n-3}w_{n-2}} \in \Gamma_2$. So, add the rule:

$$\begin{aligned}
r_t &:= (\xi_{w_1 w_2}\xi_{w_3}\xi_{w_4}\cdots\xi_{w_{n-2}}\xi_{w_{n-1}w_n}w,\\
&\qquad \xi_{w_1 w_2}\xi_{w_3 w_4\cdots w_{n-2}}\xi_{w_{n-1}w_n}w).
\end{aligned}$$

3.14 Conclusion

In this chapter we have proved that Church-Rosser languages can be described by CRLSs whose rewriting systems very much look like the production sets of context-sensitive grammars. On the one hand, this is a result which we need for our construction of prefix systems in the next chapter. Insofar, this is a technical result.

But, on the other hand, this result *characterizes deterministic growing context-sensitive languages* and can be transfered to the more general

case of GCSL, which is proved in [NW01a]. Basically, the method of weight-spreading leads to a very similar construction for growing context-sensitive grammars. In fact, the generalization to GCSL is even simpler than the deterministic case.

Yet, this alone would not justify to call the result a characterization theorem. But by proving that GCSLs can be described by weight-increasing context-sensitive grammars, one can conclude that ACSL and GCSL coincide. This is rather a surprising result: Although it relatively easy to show that ACSL $\subseteq GCSL$ (e.g. compare [Bun96]), the reverse was not expected. All in all, this shows the relevance of our normal form theorem as a characterization of DGCSL.

Chapter 4

Prefix systems

One of the properties a language class needs for practical applications is that its prefixes can be accepted by an algorithm, preferably a deterministic one. For instance, that is needed in order to detect errors and report them to a user. Unfortunately, this is not possible for the whole class of CRL. They are a basis for the recursively enumerable languages, as shown by Otto, Katsura, and Kobayashi [OKK97], which implies that there are CRLs whose prefix language are not decidable, and therefore not in CRL, either.

On the other hand, CRL is a language class with some properties which make them very interesting for use in practical applications. First of all, their word problem is decidable in deterministic linear time. Secondly, they properly contain DCFL [MNO88], see theorem 2.18 and the next chapter. Finally their mode of definition is more intuitive than that of DCFL, especially when one compares it to LR(k)-grammars.

So, instead of requiring the impossible—that CRL should be closed against building the prefix language—we use a different approach. In this chapter we give a construction for prefix systems of CRLSs. These prefix systems again are CRLSs. Considering what we said above, this construction cannot always be correct. So we have to check correctness in some way, which is also discussed in this chapter.

In the first section, we define of the *prefix construction* we are going to use. This construction leads to so-called *prefix systems*. In the second section, a formal (and quite obvious) definition for the correctness of the systems achieved that way is introduced, and we provide some examples for incorrect prefix systems. For technical reasons it is helpful to reduce right-hand sides of the rules of prefix systems. This is examined in the third section of this chapter. In the fourth section we show the easy part of correctness: the languages defined by our prefix systems always *contain* the correct prefix language.

An enhanced type of CRLSs, which we call *completion prefix systems.* is introduced in the fifth section These preserve information which otherwise would be lost be the prefix construction and which can be used for testing correctness.[1] Furthermore, if a prefix of a correct word is accepted by a prefix system, it is also possible to expand the prefix to the full word with that information. A first property which guarantees correctness of a completion prefix system is discussed in the sixth section. Because this is somewhat restrictive, we use some further methods to describe properties of completion prefix systems and of reductions possible with them. These are introduced in the seventh section. In the eighth section some relations (between rules) and data structures are defined which allow a more abstract view on reductions. With these it is possible to describe a more powerful, and decidable, property that ensures correctness. Some of the problems concerning correctness arise because CRLSs distinguish between terminals and nonterminals. This is examined in the tenth section. We end this chapter with some further thoughts about the undecidability of the correctness of prefix systems.

First drafts about prefix systems appeared in the report [Woi00d] and in [Woi00c]. Especially, the results of the ninth and tenth section are important extensions of these early versions.

4.1 The prefix construction

The idea of constructing a prefix CRLS to a csCRLS is very basic: Simply cut off suffixes of rules. To be precise, some technical efforts are necessary to handle the right-hand end of words. Given any unique definition of context-splittings, a prefix system is defined as follows:

Definition 4.1. *Let C be a csCRLS, $R = \textsc{Rules}(C)$, and $r \in R$ with context-splitting (u, v, w, x), and*

$$v = a_1 \cdots a_i \cdots a_{|v|}, a_i \in \Gamma, w = b_1 \cdots b_i \cdots b_{|w|}, b_i \in \Gamma.$$

The prefix rules of r ($\textsc{Pref}(r)$) *are defined as:*

[1] We use the term *"testing"* in order to distinguish it from *deciding* correctness.

$$\begin{aligned}\textsc{Pref}(r) := (\ & \{(uvb_1\cdots b_j\$, uxb_1\cdots b_j\$) | 0 \le j < |w|\} \\ & \cup \begin{cases} \{(a_1\cdots a_j\$, Y) | 0 < j < |v|\} & x = Y \\ \{(ua_1\cdots a_j\$, ux\$) | 0 < j < |v|\} & \textit{else} \end{cases} \\ &) \\ & \setminus \{(w,w) | w \in \Gamma^*\}.\end{aligned}$$

Define the following rewriting system R':

$$R' := R \cup \bigcup_{r \in R} \textsc{Pref}(r).$$

If R' is weight-reducing and confluent (which is decidable[2]*) then the CRLS $C' = (\Gamma, \Sigma, R', ¢, \$, y)$ is called the* prefix system of C, $C' = \textsc{Pref}(C)$.

We also use $R' = \textsc{Pref}(R)$ in that case and call R the origin *of R'.*

If R' is not confluent or not weight-reducing, $\textsc{Pref}(R)$ and $\textsc{Pref}(C)$ are not defined.

The process of building $\textsc{Pref}(R)$ *is called* prefix construction.

Remark 4.1. Each rule $r \in R$ containing a \$ appears in its own set of prefix rules: $r \in \textsc{Pref}(r)$. This will be of further importance later.

Remark 4.2. We use the trick of "undefining" $\textsc{Pref}(R)$ and $\textsc{Pref}(C)$ when the prefix construction does not produce a weight-reducing and confluent system, because in our investigations we only want to discuss csCRLSs for which the construction delivers a CRLS.

4.2 Correctness of prefix systems

Definition 4.2. *Let C be a csCRLS and assume $C' = \textsc{Pref}(C)$ is defined. C' is a* correct prefix system *if and only if $L_{C'} = Pref(L_C)$.*

[2] Weight-reduction can be decided by solving a system of inequations, and if the rewriting system is weight-reducing, confluence can be tested with the ConfTest-algorithm.

Theorem 4.1. *There are csCRLS C such that* $\textsc{Pref}(C)$ *is not defined or not correct (i.e., $L_{C'} \neq Pref(L_C)$).*

Proof. We consider three csCRLSs C_1, C_2, C_3.

Let $\Sigma = \{a, b, c\}$ and $\Gamma = \{a, b, c, D, ¢, \$, Y\}$ be the common alphabets of the three systems.

Assume R'_1, R'_2, R'_3 would be the result of the prefix construction if defined.

1. $R_1 = \{(a, b), (ba, a), (bb, a), (¢a\$, Y), (¢b\$, Y)\}$
   $R'_1 = R_1 \cup \{(b\$, a\$), (¢\$, Y)\}$
   R'_1 is not weight-reducing system, in fact, it is not terminating:
   $a\$ \underset{R'_1}{\longrightarrow} b\$ \underset{R'_1}{\longrightarrow} a\$$.
   Therefore, $\textsc{Pref}(R_1)$ is not defined.
2. $R_2 = \{abb\$, a\$), (bbc, d), (¢a\$, Y)$
   $R'_2 = R_2 \cup \{ab\$, a\$), (bb\$, d\$), (b\$, d\$), (¢\$, Y)\}$
   R'_2 is not confluent:
   $¢abb\$ \underset{R'_2}{\longrightarrow}^* Y \in \textsc{Irr}(R'_2)$ and $¢abb\$ \underset{R'_2}{\longrightarrow}^* ¢ad\$ \in \textsc{Irr}(R'_2)$.
   Therefore, $\textsc{Pref}(R_2)$ is not defined.
3. $R_3 = \{(¢D\$, Y)\}$
   $R'_3 = R_3 \cup \{(¢\$, Y)\}$
   $L_{C_3} = \emptyset$ but $L_{C'_3} = \{\Box\}$.
   Therefore, $\textsc{Pref}(R_3)$ is not correct. $\Box$

4.3 Reducing right-hand sides

Counter example 1 of the proof of theorem 4.1 needs some closer examination. We observe that R_1 is not normalized. Normalizing it leads to the following rewriting system:

$$R_n = \{(a, b), (bb, b), (¢b\$, Y)\}.$$

Then $\textsc{Pref}(R_n)$ is defined:

$$\textsc{Pref}(R_n) = \{(a, b), (bb, b), (¢b\$, Y), (¢\$, Y)\}.$$

Why is it not possible to simply require normalization before applying the prefix construction? This would give a bigger class of CRLSs which have a well defined prefix system. Consider the following:

Example 4.1. Let $R = \{(ab, ad), (ad, cd), (¢cd\$, Y)\}$ be the rewriting system of a csCRLS C. Normalizing R gives the rewriting system

$$R' = \{(ad, cd), (ab, cd), (¢cd\$, Y)\}.$$

But the rule (ab, cd) is not context-splittable. So we cannot apply the prefix construction after normalization.

The situation is not as problematic as this example might imply:

Example 4.2. We use the above counter example again. But instead of normalizing it we reduce the right-hand sides of the (new) prefix rules with the rules of R_1 and remove all rules which have identical left and right-hand sides after this. In fact, we only need to use the rule (a, b) and we apply it on the rule $(b\$, a\$)$. The alternative rewriting system obtained this way is:

$$R''_1 = R_1 \cup \{(¢\$, Y)\}.$$

R'_1 and R''_2 are equivalent w.r.t. to the Thue congruence: $\underset{R'_1}{\longleftrightarrow}^* = \underset{R''_2}{\longleftrightarrow}^*$. But the latter is weight-reducing and the former is not.

In order to utilize this idea, we define a variant of prefix systems. Since we do not want to lose too much of the information about the original system, we only use those old rules that do not work at the right-hand side of a word, i.e., they contain no \$ letter. Extracting this subset of rules will be denoted the following way:

Definition 4.3. *Let R be a rewriting system defined on the alphabet Γ, and $\alpha \in \Gamma$. Then $R \backslash \alpha$ is the subsystem of R that is obtained by removing all rules containing the letter α:*

$$R \setminus \alpha := R \cap (\Gamma \setminus \alpha) \times (\Gamma \setminus \alpha).$$

Before we are going to use this, we have to consider whether after extracting the rules we will still have a weight-reducing and confluent rewriting system.

Lemma 4.1. *Let R be a (weight-reducing and) confluent rewriting system defined on the alphabet Γ, and $\alpha \in \Gamma$. Assume for every rule $(u, v) \in R$ the letter α appears in v only if it appears in u. Then $R' = R \setminus \alpha$ is (weight-reducing and) confluent.*

Proof. Since weight-reduction is a local property of each rule it is directly transfered. Global confluence implies local confluence, and therefore we directly prove the former. Assume w_1 and w_2 are a critical pair of R', that is there exists a $w \in (\Gamma \setminus \alpha)^*$ such that $w \underset{R'}{\longrightarrow}^* w_1$, $w \underset{R'}{\longrightarrow}^* w_2$, and $w_1 \neq w_2$. We know that there exists a w_3 such that $w_1 \underset{R}{\longrightarrow}^* w_3$ and $w_2 \underset{R}{\longrightarrow}^* w_3$.
Because R does not add an α to a word that does not already contain one, we know:

$$\forall w \in (\Gamma \setminus \alpha)^*, w', w'' \in \Gamma^*, r \in R : w \underset{R}{\longrightarrow}^* w' \underset{r}{\longrightarrow} w''$$
$$\Longrightarrow$$
$$w', w'' \in (\Gamma \setminus \alpha)^* \wedge r \in R'.$$

That means, every descendant of w in R again is in $(\Gamma \setminus \alpha)^*$ and each rule applicable to w or any of its descendants does not contain an α and therefore is in R', too. Then $w_1 \underset{R'}{\longrightarrow}^* w_3$ and $w_2 \underset{R'}{\longrightarrow}^* w_3$ holds, and so R' is confluent. □

Note that it is crucial that α is not newly introduced into a word by any rule of R. For a csCRLS C this at least holds for the end marker letters. Furthermore, we wish to emphasize the fact that the proof of confluence does not depend on weight-reduction.

Definition 4.4. *Let C be a csCRLS, $R = \textsc{Rules}(C)$, and $r \in R$ with context-splitting (u, v, w, x), and*

$$v = a_1 \cdots a_i \cdots a_{|v|}, a_i \in \Gamma, w = b_1 \cdots b_i \cdots b_{|w|}, b_i \in \Gamma.$$

The reduced prefix rules of r *(*$\text{RPREF}(r)$*) are defined as:*

$$\begin{aligned}\text{RPREF}(r) := (\ & \{(uvb_1 \cdots b_j\$, [uxb_1 \cdots b_j\$]_{R\setminus\$}) | 0 \leq j < |w|\} \\ & \cup \begin{cases} \{(a_1 \cdots a_j\$, y) | 0 < j < |v|\} & x = y \\ \{(ua_1 \cdots a_j\$, [ux\$]_{R\setminus\$}) | 0 < j < |v|\} & else \end{cases} \\ &) \\ & \setminus \{(w, w) | w \in \Gamma^*\}\end{aligned}$$

Define the following rewriting system R'*:*

$$R' = R \setminus \$ \cup \bigcup_{r \in R} \text{RPREF}(r).$$

If R' *is weight-reducing and confluent (which is decidable) then the CRLS* $C' = (\Gamma, \Sigma, R', ¢, \$, y)$ *is called* reduced prefix system of C, $C' = \text{RPREF}(C)$.

Similar to prefix system, we also use $R' = \text{RPREF}(R)$ *(and* $\text{RPREF}(C)$*) in that case and call* R *the* origin *of* R'*. If* R' *is not confluent or not weight-reducing,* $\text{RPREF}(R)$ *and* $\text{RPREF}(C)$ *are not defined.*[3]

Remark 4.3. With remark 4.1 we see that the use of $R \setminus \$$ instead of R in the definition of $\text{RPREF}(R)$ prevents ambiguity but does not change the accepted language of $\text{RPREF}(C)$ if $\text{RPREF}(R)$ is defined (because of the required confluence).

Remark 4.4. The original prefix construction, as in definition 4.1, is not defined for every csCRLS C, because it may produce a rewriting system which is not weight-reducing or not confluent. In consequence, there exist CRLSs C such that $\text{PREF}(C)$ is not defined, but $\text{RPREF}(C)$ is defined. See also example 4.2.

Definition 4.5. *Let* C *be a csCRLS and assume* $C' = \text{RPREF}(C)$ *is defined.* C' *is* correct *if* $L_{C'} = Pref(L_C)$.

[3] Again, as stated in remark 4.2, $\text{RPREF}(C)$ is either a CRLS or not defined.

4.4 The prefix language is included

Theorem 4.2. *Let C be a csCRLS and $C' = \textsc{Pref}(C)$ the prefix system with R' as rewriting system of C'. Then Pref$(L_C) \subseteq L_{C'}$.*

Remark 4.5. We are not going to expect any special context-splitting. It is only required that for each rule in the original system one such splitting was used to construct its set of prefix rules. If we had a definition of context-splittings which would identify a unique splitting for any rule, one could use this. On the other hand, it might be useful to choose a splitting for each rule with regard to other criteria (e.g. size or number of prefix rules). Since the following proof does not depend on the splitting, this would be justified, too.

Proof. We prove a stronger property:

Claim.

$$\forall n > 0, s, t \in \Gamma^*_{\text{inner}} : ¢st\$ \underset{C}{\longrightarrow}^n Y \Longrightarrow ¢s\$ \underset{C'}{\longrightarrow}^* Y.$$

We use an induction on the length of the reduction in C.

Induction basis. (n=1) Follows directly from the construction of terminating rules in $\textsc{Pref}(C)$.

Induction claim. Suppose, the claim holds for $n \geq 1$:

$$\forall s', t' \in \Gamma^*_{\text{inner}} : ¢s't'\$ \underset{C}{\longrightarrow}^n Y \Longrightarrow ¢s'\$ \underset{C'}{\longrightarrow}^* Y.$$

Induction step. $(n \to n+1)$
Let the reduction $st = w_1w_2w_3 \in \Gamma^*_{\text{inner}}, x \in \Gamma_{\text{inner}} \cup \{\Box\}$, and $r = (uvw, uxw) \in \textsc{Rules}(C)$ be given with (u, v, w, x) being a context-splitting of r with $v = w_2$. Let the following hold:

$$¢st\$ = ¢w_1w_2w_3\$ \underset{r}{\longrightarrow} ¢w_1xw_3\$ \underset{C}{\longrightarrow}^n Y.$$

There are four cases. In the following, "PC" means the reduction is possible because of the properties of the prefix construction for C' and "IC" means "by induction claim."

Case 1: s is prefix of w_1 (i.e., u is a true suffix of or equal to w_1): this case is equivalent to the claim for n.

Case 2: s is prefix of w_1w_2: $¢s\$ \xrightarrow[C']{\text{PC}} ¢w_1x\$ \xrightarrow[C']{\text{IC}}{}^{*} Y$.

Case 3: $s = w_1w_2w_3'$ with w_3' is true prefix of w_3:
$¢w_1w_2w_3'\$ \xrightarrow[C']{\text{PC}} ¢w_1xw_3'\$ \xrightarrow[C']{\text{IC}}{}^{*} Y$.

Case 4: $s = w_1w_2w_3$: Then the rule is in C. Then it is also in C', so the claim directly is true.

Figure 4.1 illustrates the first three cases. □

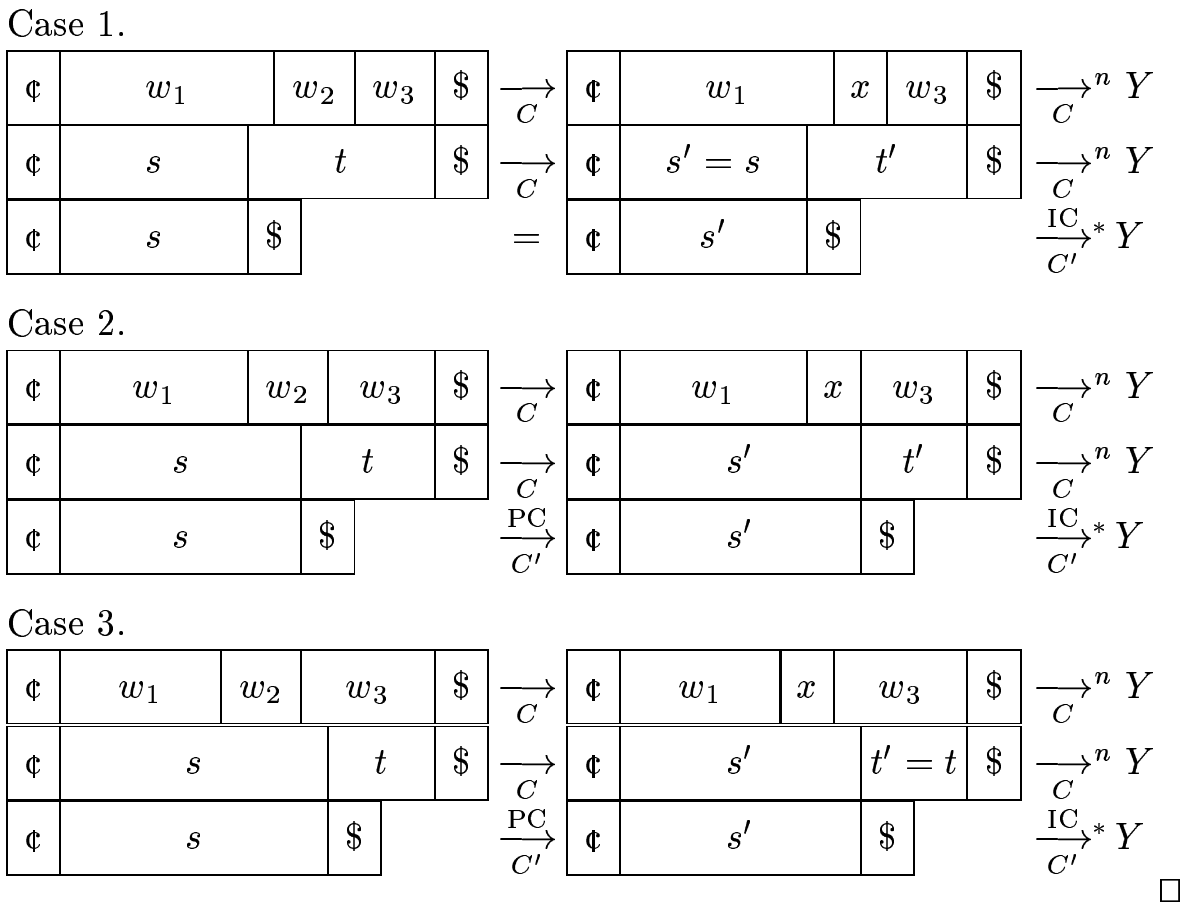

Fig. 4.1: Illustration of the proof of theorem 4.2

Corollary 4.1. *Let C be a csCRLS and assume $C' = \text{RPREF}(C)$ is defined with R' as rewriting system of C'. Then Pref$(L_C) \subseteq L_{C'}$*

Proof. There is a $\varphi : \Gamma^* \to \mathbb{N}$ which is a weight function compatible with R. The proof is essentially the same as for theorem 4.2. Instead of an induction on the length of the reduction, an induction on the weight of $¢s't'\$$ has to be used. The induction basis is adapted, accordingly. The induction step again argues with the first rewriting step and the weight reduction from n to $n' < n$. In cases 2 to 4 of the case distinction one takes the confluence of R and the rules of R implicitly used by rules of R' into account. Note that case 4 only differs from the proof of theorem 4.2 if the rules work on the right-hand side of a word. □

4.5 Completion prefix systems

For further inspection of the correctness of prefix systems, we want to keep the information about what has been cut off by the prefix construction. For that purpose, we introduce the following:

Definition 4.6. *Let C be a csCRLS with rewriting system R and $r \in R$ with context-splitting (u, v, w, x), $v = a_1 \cdot a_2 \cdots a_i \cdots a_{|v|}$, $a_i \in \Gamma$, $w = b_1 \cdot b_2 \cdots b_i \cdots b_{|w|}$, $b_i \in \Gamma$. If* $\mathrm{RPref}(C)$ *is defined, we define the set* $\mathrm{CPref}(r)$ *of* completion prefix rules of r *(or, shorter, completion rules):*

$$
\begin{aligned}
\mathrm{CPref}(r) = (\ & \{(uvb_1 \cdots b_j\$, [uxb_1 \cdots b_j\$]_{R\setminus\$}, \Box, b_{j+1} \cdots b_{|w|}) | 0 \le j < |w|\} \\
& \cup \begin{cases} \{(a_1 \cdots a_j\$, y, a_{j+1} \cdots a_{|v|-1}, \$) | 0 < j < |v|\} & x = Y \\ \{(ua_1 \cdots a_j\$, [ux\$]_{R\setminus\$}, a_{j+1} \cdots a_{|v|}, w) | 0 < j < |v|\} & x \neq Y \end{cases} \\
& \cup \{(uvw, uxw, \Box, \Box) | vw \notin \Gamma^* \cdot \$\}^4 \\
&) \\
& \setminus \{(w', w', u', v') | w', u', v' \in \Gamma^*\}.
\end{aligned}
$$

– The third component of the completion rules is called *consumed completion.*

[4] This set is empty or a singleton.

- The fourth component is called *unconsumed completion.*
- The system R' is called *completion prefix system of* R with $R' = \bigcup_{r \in R} \text{CPREF}(r)$. With $\text{REW}(R')$ or $\text{REW}(C')$ we denote the extraction of the first two components of each completion rule, which has a rewriting system as result.
- In the same manner, CPREF will be used for the completion prefix CRLS given by the alphabets and accepting letter of C and R' if and only if $\text{RPREF}(C)$ is defined.
- $\text{CRULES}(R') := R' \setminus \{(u, v, \Box, \Box) | (u, v) \in \Sigma^* \times \Sigma^*\}$ is the set of rules from R' with nonempty completions.
- The set of all (old) rules not applicable at the right-hand end can be identified with $R' \setminus \$ = R' \setminus \text{CRULES}(R')$.
- $\text{CPREF}(C')$ is the completion prefix which is given by the expanded rewriting system $\text{CPREF}(R')$ and the remaining parts being identical to those of C'.
- $C' \setminus \$$ is defined accordingly.

We expand the notion of rewriting systems to completion systems:

Definition 4.7. *Let C be a csCRLS and assume $C' = \text{CPREF}(C)$ is defined and has the set of completion rules R'. Then the rewriting relation for R' is defined as:*

$$\underset{R'}{\longrightarrow} := \{(xuy, xvy) | x, y \in \Gamma^*, (u, v) \in \text{REW}(R')\}.$$

Accordingly, $\underset{C'}{\longrightarrow} := \underset{R'}{\longrightarrow}$.

Let $R'' = \text{REW}(\text{RPREF}(C))$. By construction, $\underset{R'}{\longrightarrow}$ is identical to $\underset{R''}{\longrightarrow}$. The notions of the language of C' and of correctness of C' are transfered in the natural way from CRLSs and $\text{RPREF}(C)$, respectively.

4.6 When is every accepted word a correct prefix?

Now we look at completion prefix systems and show how they can be used to check if a prefix system is correct. First, we provide an example csCRLS with an explicitly given splitting. For this, we will build the completion prefix system:

Example 4.3. The csCRLS C is defined in the following way: Let $\Sigma = \{a, b, c, d, e\}$, $\Gamma = \Sigma \cup \{\$, ¢, Y\}$ with left- and right-hand end markers ¢ and \$ and accepting symbol Y. Let the rewriting system R be defined as follows. We mark a context-splitting[5] with concatenation dots '·', these dots do not belong to the rules:

$$\begin{aligned} \cdot abc\cdot &\to \cdot b\cdot \\ \cdot abb\cdot bc &\to \cdot a\cdot bc \\ \cdot bb\cdot \$ &\to \cdot b\cdot \$ \\ \cdot db\cdot &\to \cdot b\cdot \\ \cdot e\cdot c &\to \cdot b\cdot c \\ \cdot ¢b\$\cdot &\to \cdot Y\cdot \end{aligned}$$

In figure 4.2 the completion prefix system $C' = \text{CPREF}(C)$ is given.

Now we apply the prefix system of our example to an input word. Considering only the Thue part of the rules, we look at a prefix of the word $adabbdecc \in L_C$ and its acceptance in C'. Observe the mixture of rules already in R and those new rules from $R' \setminus R$:

$$¢adabbde\$ \underset{r_{12}}{\to} ¢adabbdb\$ \underset{r_9}{\to} ¢adabbb\$ \underset{r_5}{\to} ¢adab\$ \underset{r_2}{\to} ¢adb\$ \underset{r_9}{\to} ¢ab\$ \underset{r_2}{\to} ¢b\$ \underset{r_{13}}{\to} Y.$$

This seems to work fine, but we cannot always be sure that a prefix system is doing what we would expect it to do.

The meaning of the brackets in figure 4.2 are as follows: The words left of the slashes are consumed completions. They will be of no importance for further reductions. On the right-hand side of the slashes are the unconsumed completions. They play an important role for the correctness of prefix systems: they link applications of prefix rules. To explain this, we have a closer look at a part of the above reduction. Below the $\to$ we write the bracket expression of the respective rules:

$$¢adabbde\$ \underset{[\square/c]}{\longrightarrow} ¢adabbdb\$ \underset{[\square/\square]}{\longrightarrow} ¢adabbb\$ \underset{[\square/c]}{\longrightarrow} ¢adab\$ \underset{[c/\square]}{\longrightarrow} ¢adb\$.$$

[5] Note that this choice is not necessarily optimal.

r_1	abc	$\to b$	$[\square/\square]$
r_2	$ab\$$	$\to b\$$	$[c/\square]$
r_3	$a\$$	$\to b\$$	$[bc/\square]$
r_4	$abbbc$	$\to abc$	$[\square/\square]$
r_5	$abbb\$$	$\to ab\$$	$[\square/c]$
r_6	$abb\$$	$\to a\$$	$[\square/bc]$
r_7	$ab\$$	$\to a\$$	$[b/bc]$
r_8	$bb\$$	$\to b\$$	$[\square/\$]$
r_9	db	$\to b$	$[\square/\square]$
r_{10}	$d\$$	$\to b\$$	$[b/\square]$
r_{11}	ec	$\to bc$	$[\square/\square]$
r_{12}	$e\$$	$\to b\$$	$[\square/c]$
r_{13}	$¢b\$$	$\to Y$	$[\square/\$]$
r_{14}	$¢\$$	$\to Y$	$[b/\$]$

Fig. 4.2: The rules of the completion prefix system for example 4.3

The application of rule r_{12} means: "guess that the next letter would be a c and that it belonged to the unchanged right context." Then rule r_9 is used. Since this is an old rule, it does not change the guess of a completion. After that, rule r_5 is used. Again, it assumes an unchanged c to be the next letter of a possible completion to a correct word. The application of rule r_5 fits to that of r_{12} which assumed the same completion. Now rule r_2 is used. Here the c is still the same letter but is part of the consumed completion. This means in other words: We guessed the next letter to be a c, this guess was correct, and now it is completely used, so we do not need to consider it further.

In contrast to this, consider using rule r_8 twice instead of r_5. This rule guesses the end of the word. So its unconsumed completion will not fit to the completion of r_2. This is of major importance. Because of r_8, $L_{C'}$ would also contain $abbbb$ which clearly is not a prefix of a word in L_C.

Formalizing this observations leads to the following definition:

Definition 4.8. *Let $C = (\Gamma, \Sigma, R, ¢, \$, Y)$ be a csCRLS and $C' = \text{CPREF}(C)$, with R' being the rules of C'. Let $w, w' \in \Gamma^*$. Let $s \in R'^*$ be a sequence $s = r_1 r_2 \cdots r_n$ of rules with length n such that:*

$$w \underset{r_1}{\longrightarrow} w_1 \underset{r_2}{\longrightarrow} w_2 \underset{r_3}{\longrightarrow} \cdots \underset{r_n}{\longrightarrow} w'.$$

Definition 4.9. *We apply* CRULES *to sequences of (completion) rules in the following way Let* $\textsc{Crules} : R'^* \to \textsc{Crules}(R')^*$ *be the morphism defined by its result on single elements of* R' *by:*

$$\textsc{Crules}(x) := \begin{cases} x & \text{if } x \in \textsc{Crules}(R') \\ \square & \text{otherwise.} \end{cases}$$

Since it always will be clear whether we speak about sequences of reduction rules or we do not, this double use should cause no irritations.

Let $\textsc{Crules}(s) = s' = r'_1 \cdot r'_2 \cdots r'_m$. *Then* s *is a* valid reduction *if for all* $i, (1 \leq i < m)$ *the unconsumed completion of* r'_i *is a prefix of the concatenation of the consumed and the unconsumed completion of* r'_{i+1}.

Furthermore, s *is a valid reduction with* finished completion *(or, shortly, a* finished valid reduction*), if the unconsumed completion* c_u *of* r'_m *is empty or if* $c_u = \$$ *and the result* w' *of the reduction is* Y.

Finally, s *is a* basic valid reduction *if it is a finished valid reduction and there is no* $i < m$ *such that* r'_i *has an empty unconsumed completion.*

Similarly to the double use CRULES, the function REW which extracts the rewriting rules from a completion prefix system is extended to work on sequences of completion rules. In doing this, we are able to speak of the rewriting part of completion rules separately from the completion part. Since the problem of correctness of prefix systems (without completion information) is exactly hidden behind this abstraction, we want to know what information is gained by using completion prefix systems instead. Or alternatively, we can ask how we can map the latter into the former. Using the notion of mappings now leads to asking questions about surjectivity. This is the next step in our discussion: Given an accepting reduction in a prefix system, can we map it to a valid reduction with finished completion in the corresponding completion prefix system? This leads to a first theorem about correctness:

Theorem 4.3. *Let* $C = (\Gamma, \Sigma, R, ¢, \$, Y)$ *be a csCRLS with* $\Gamma = \Sigma \cup \{¢, \$, Y\}$, $C' = \text{RPREF}(C)$, *and* $C'' = \text{CPREF}(C)$. *Let* R, R', *and* R'' *be the respective rule sets.*

For every $w \in \Sigma^*$, $¢w\$ \underset{s' \in R'^*}{\longrightarrow}{}^* Y$, *where* s' *is a sequence of rules in* R' *such that there exists a valid reduction* $s'' \in R''^*$ *with finished completion, and* $\textsc{Rew}(s'') = s'$, *there exist* $c \in \Sigma^*$ *and* $s \in R^*$ *such that* $¢wc\$ \underset{s}{\longrightarrow}{}^* Y$ *and* $|s| \geq |s'|$.

Proof. We prove the theorem by an induction over the number of reduction steps, denoted with n.

Induction basis. (n=1) Let $(u, Y) = r'_1$. Since r'_1 is an accepting rule and s'' is valid, then by the properties of the prefix construction there is a c_c such that $r''_1 = (u, Y, c_c, \$)$. There is a u' such that $u = ¢u'\$$.

Because of the construction of R' and R'' this means $(¢u'c_c\$, Y) \in R$. Then the sequence s we are looking for is $s = (¢u'c_c\$, Y)$ and $|s| = |s'| = |s''| = 1$. The completion c is c_c. Note that c can be the empty word.

Induction claim. Assume the theorem holds for $n \geq 1$.

Induction step. $(n \to n+1)$
We may assume $s' = r'_0 r'_1 \cdots r'_n$ and $s'' = r''_0 r''_1 \cdots r''_n$ with $\textsc{Rew}(s'') = s'$. Furthermore s'' is a valid reduction with finished completion.

Then there is a w' such that $¢w\$ \underset{r'_0}{\longrightarrow} ¢w'\$ \underset{r'_1 \cdots r'_n}{\longrightarrow}{}^* Y$. By the induction claim there exist:

1. $t = r_1 \cdots r_m \in R^*$ with $m \geq n$ and
2. $c_o \in \Sigma^*$ with $¢w'c_o\$ \underset{t}{\longrightarrow}{}^* Y$.

Since s'' is valid, there exists an i such that $1 \leq i \leq n$ and r''_i is the first completion rule with index greater than 1 in the reduction that has nonempty completions. There are u, v such that $r'_0 = (u, v)$.

There are three cases:

1. $r'_0 \in R \setminus \$$. Then $r'_0 \in R$ and we know $r''_0 = (u, v, \Box, \Box)$. Therefore $r''_0 \in R''$ and $\textsc{Rew}(r''_0) = r'_0$ and:

   $$¢w\$ \underset{r'_0}{\longrightarrow} ¢w'\$ \underset{t}{\longrightarrow}{}^* Y.$$

   In consequence $r_0 = r'_0$ can be chosen, the completion is $c = c_o$ (it is unchanged), and $s = r_0 t$ is the rule sequence in R we are looking for. The length of s is also correct because $m + 1 \geq n + 1$.

2. $r_0' \notin R \setminus \$$ and $r_0'' = (u\$, v\$, c_c, \Box)$ with $c_c \neq \Box$. Because the unconsumed completion of r_0'' is empty the rest of s'' without r_0'' is valid, it directly follows that c_c fits to the completion. By the construction of R' and R'' we know there exists a sequence t' of reductions in R such that $t' = t_1 t_2 \cdots t_l \in (R \setminus \$)^+$, $t_1 = (uc_c, v) \in R$ and:

$$¢wc_c c_o\$ \underset{t'}{\longrightarrow}^* ¢w'c_o\$ \underset{t}{\longrightarrow}^* Y.$$

   Therefore the new completion is $c = c_c c_o$, the new rule sequence in R is $s = t't$, and $|s| \geq n + 1$. Note that the fact that $l \geq 1$ reflects the full reduction of right-hand prefix rule sides with $R \setminus \$$.
3. $r_0' \notin R \setminus \$$ and $r_0'' = (u\$, v\$, c_c, c_u)$ with $c_u \neq \Box$. Because s'' is valid we can assume that there are u', v', c_c', and c_u' with $r_1'' = (u', v', c_c', c_u')$ and $c_u \in \mathrm{Pref}(c_c' c_u')$. Also $c_c' c_u' \in \mathrm{Pref}(c_o)$ and in consequence c_u is a prefix of c_o.
   By the construction of R' and R'' we know there exists a sequence t' of reductions in R such that $t' = t_1 t_2 \cdots t_l \in R \cdot (R \setminus \$)^+$, $t_1 = (uc_c c_u, vc_u) \in R$, and:

$$¢wc_c c_o\$ \underset{t'}{\longrightarrow}^* ¢w'c_o\$ \underset{t}{\longrightarrow}^* Y.$$

   Therefore the completion is $c = c_c c_o$, the new rule sequence in R is $s = t't$, and $|s| \geq n + 1$. $\Box$

A look back at example 4.3 shows that the correct reduction with the prefix system exactly provides the conditions of this theorem. On the other hand, the example of the incorrect reduction reveals a problem: There may be reductions that are not valid—and therefore not providing a correct completion—that nevertheless accept correct prefixes. A close look at the completion prefix system shows that other rules of this system, beside r_8, lead to even more words accepted which are no prefixes. Certainly, we would like to have an algorithm that finds all those cases. In the rest of this chapter we will discuss up to which extent this is possible.

4.7 Relevant parts of completion prefix reductions

One problem concerning the necessary conditions of theorem 4.3 is that they only show correctness of single reductions. Since usually it will not

be possible to inspect all reductions we have to find a way to overcome this restriction.

The first question, which will be answered in this section, is how much of the reductions really is important for correctness. There are four observations:

1. First, the proof of the theorem shows that rules from a completion prefix system that do not work at the right-hand end of a word do not change the completions required.
2. In order to check whether a completion prefix reduction s is valid or not, we only need to look at $\textsc{Crules}(s)$.
3. The proof of the theorem does not use the fact that the rewriting systems in discussion always are confluent.
4. Whenever a rule uses up the full completion, the next completion rule in the reduction may have any completion. It only has to fulfill the conditions of the rewriting relation. Because of this we already introduced the notion of basic valid reductions with finished reductions. They provide a kind of "smallest parts" of reductions that have to be examined.

Now we will gather these observations more formally in a series of propositions. They all have the same common requirements. For propositions 4.1 to 4.3 assume the following:

Let $C = (\Gamma, \Sigma, R, ¢, \$, Y)$ be a csCRLS with $\Gamma = \Sigma \cup \{¢, \$, Y\}$ and $C' = \textsc{CPref}(C)$. Let R and R' be the respective rule sets.

Proposition 4.1. *Let $r = (u\$, v\$, c_c, c_u) \in R'$. Then:*

1. *For all $w \in \Gamma^*$: $wu\$ \underset{r}{\longrightarrow} wv\$$.*
2. *There do not exist $w \in \Sigma^*$ and $w', w'' \in \Gamma^*$ such that $w'' \neq \square$ and $¢w\$ \underset{R'}{\longrightarrow}^* w'u\$w''$.*
3. *For all $w \in \Gamma^*$: If $w \notin ¢\Gamma^*\$$ then $w \underset{R'}{\not\longrightarrow}^* Y$.*

Proof. All properties follow directly from the definitions of csCRLS and $\textsc{CPref}$. $\square$

Proposition 4.2. *Let* $s \in (R' \setminus \$)^* \cdot \textsc{Crules}(R') \cdot (R' \setminus \$)^*$ *with* $s = r_1 r_2 \cdots r_i \cdots r_n$ *where* r_i *is the single completion rule in* s*:* $r_i \in \textsc{Crules}(R')$*. Assume* $r_i = (uv\$, uv'\$, c_c, c_u)$*, i.e.* u *is the unchanged prefix of the rewriting part of* r_i*. Assume*

$$wu \underset{r_1}{\longrightarrow} w_1 u \underset{r_2}{\longrightarrow} w_2 u \underset{r_3}{\longrightarrow} \cdots w_{i-2} u \underset{r_{i-1}}{\longrightarrow} w_{i-1} u$$
$$\text{and}$$
$$w_{i+1} u \underset{r_{i+1}}{\longrightarrow} w_{i+1} u \cdots w_{n-1} u \underset{r_n}{\longrightarrow} w_n u.$$

Then for all permutations s' *of* s *that can be produced from* s *by moving* r_i *arbitrarily far to the left or right we get the same reduction result:*

$$wuv \underset{s}{\longrightarrow}^* w_n uv' \text{ if and only if } wuv \underset{s'}{\longrightarrow}^* w_n uv'.$$

Proof. This is obvious. □

That means, as long as old rules from R that do not work at the right-hand end of the word (and which are represented by a rule in R' with empty completions) do not overlap too much with a true completion rule their exact time of application does not matter w.r.t. the completion rule. As a consequence, we may use confluence in order to sort such reductions in any desired manner.

Unfortunately, sometimes old rules *do overlap further* with completion rules. So we have to handle this, too. The following example shows the kind of problems we have to deal with:

Example 4.4. Consider a system containing the rule $r = (abbb, bbb)$. In the completion prefix system this rule appears with empty completions as $r_0 = (abbb, bbb, \square, \square)$. One of the completion prefix rules for r is $r_1 = (abb\$, bb\$, \square, b)$. Assume further a rule $r_2 = (¢bbb\$, Y, b, \$)$ is in the completion prefix system. Actually, the completion system must contain more rules, but we only need these three for our example.

The following reduction is possible:

$$¢ababb\$ \underset{r_1}{\longrightarrow}^* ¢abbb\$ \underset{r_0}{\longrightarrow}^* ¢bbb\$ \underset{r_2}{\longrightarrow}^* Y.$$

The use of rule r_0 in this example implies that r_1 has been used before. So, we cannot use the proposition above in order to sort the reduction steps. Although r_0 is an old rule with empty completions, we would like to store the completion information introduced by r_1. With the next proposition and the definition following it we introduce a construction which allows us to do this.

Definition 4.10. *Let $lcp(u, v)$ denote the length of the longest common prefix of two words $u, v \in \Gamma^*$.*

Proposition 4.3. *Let $r = (u, v, c_c, c_u) \in \textsc{Crules}(R')$, $c_u \neq \Box$, and $r' = (u', v', \Box, \Box) \in R' \setminus \$$ such that $u = u_1u_2$, $v = u_1v_1$ and $|u_1| = lcp(u, v)$. Assume $u' = u_0u_1u_3$ where $u_3 \in Pref(v_1)$ and $u_3 \neq \Box$. Let w be the word which is acquired by applying first r and then r' on the word $u_0u_1v_1$:*

$$u_0u_1u_2 \underset{r}{\longrightarrow} u_0u_1v_1 \underset{r'}{\longrightarrow} w.$$

Therefore u' overlaps with v by more than $lcp(u, v)$ and the reduction with r' at the given position[6] requires the application of r.

We can build a new completion prefix system R'' by adding the following rule r'' to R'.

$$r'' = (u_0u_1v_1, [u_0u_1v_1]_{R'\setminus\$}, \Box, c_u) \text{ and } R'' = R' \cup \{r''\}.$$

Then the new rule preserves confluence and the completion information and R'' is equivalent to R' w.r.t. the accepted words. In figure 4.3 the relations of the words are depicted. The new rule is derived from the two lines marked with $(*)$.

Proof. Since r'' only simulates a sequence of reductions with a part of R' it is clear that its addition does not change the set of accepted words. The completion information is handled correctly because only rules from $R' \setminus \$$ are used. $\Box$

[6] It is possible that r' could be applied further left, but this is handled by the last proposition.

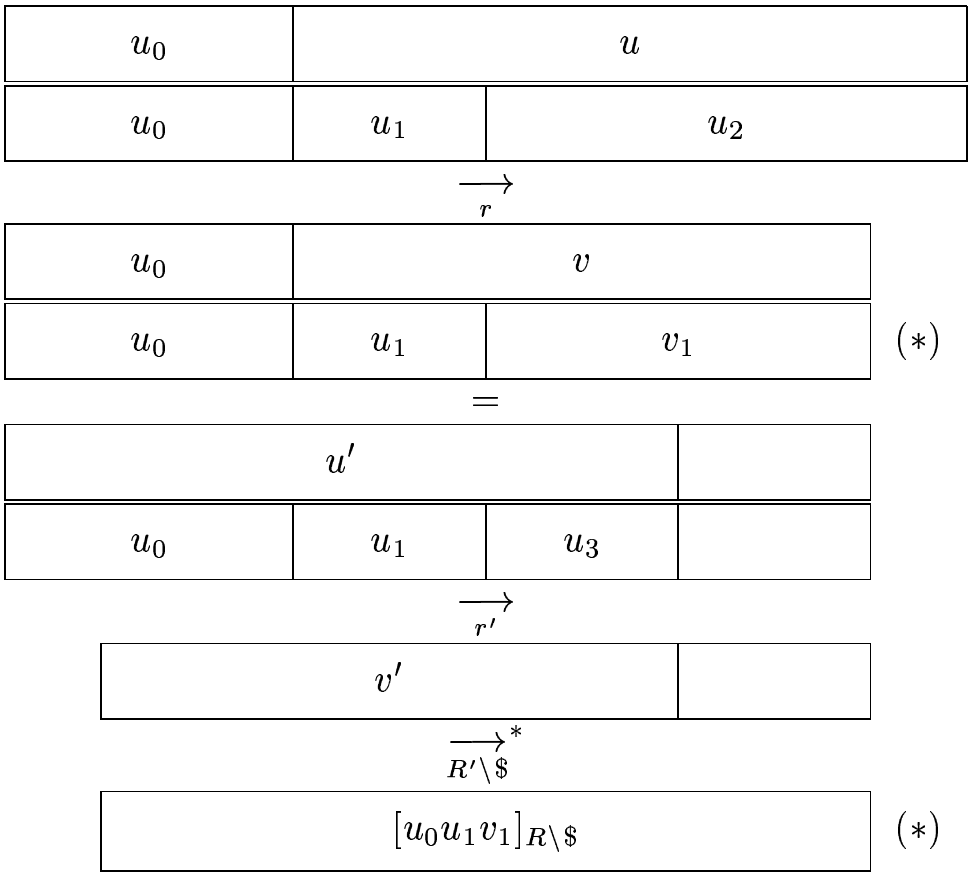

Fig. 4.3: Construction of new candidates

Note that this simulation does not start with r and only uses old rules from $R' \setminus \$$. The new rule is not used to combine the effects of r and r', but only to allow an equivalent reduction starting with r' linked to the right hand side of a word which also stores completion information.

Furthermore, we can repeat this procedure with R'' as new input and for every possible rule combination. Either it is possible to repeat this infinitely often, always finding a new rule that was not already added, or this process can be stopped because no more new rules are possible.

Definition 4.11. *Let R^∞ be the limit of this process of adding new rules. Then we call $K_{R'} = \textsc{Crules}(R^\infty)$ the* candidate set of R'. *If the procedure described above can be repeated infinitely often, always adding new rules, we call $K_{R'}$ the* infinite candidate set *of R', and we denote this with $K^\infty_{R'}$. If the procedure does not add any more rules after finitely*

many steps, we call the resulting $K_{R'}$ the (finite) maximal candidate set *of R'. If we want to stress this we denote it with $K^f_{R'}$.*

Example 4.5. (Part 1 continues example 4.4)

1. Consider a system containing the rule $r = (abbb, bbb)$.
   (a) In the completion prefix system this rule appears with empty completions as $r_0 = (abbb, bbb, \square, \square)$. One of the completion prefix rules for r is $r_1 = (abb\$, bb\$, \square, b)$. Now the first component of r_0 overlaps enough with the second component of r_1, and therefore $r_2 = (abbb\$, bbb\$, \square, b)$ is a new candidate. Again, r_0 overlaps with r_2. So we get another candidate $r_3 = (abbbb\$, bbbb\$, \square)$. By induction one can show that the candidate set of a completion prefix system containing r_0 and r_1 is infinite and contains the following set of candidates: $\{(ab^n\$, b^n\$, \square, b) | n \geq 2\}$.
   (b) Ignoring confluence at the moment and assuming that there is a rule $r' = (bb, c, \square, \square)$ in the completion prefix system, too, and that $c \in \textsc{Irr}(R' \setminus \$)$. For the construction of completion prefix rules, the second component of the rules is fully reduced with $R' \setminus \$$. So instead of r_1 the completion prefix rule will be $r'_1 = (abb\$, c\$, \square, b)$. If we assume no other rule exists which again adds more candidates because it overlaps with the second component of r'_1 we do not get an infinite sequence of candidates because of r'_1.
   Of course, this second possibility is too simple an example because without any other rules r and r' would produce critical pairs that cannot be resolved to a common descendant. Furthermore, the rule r leads to more completion rules, e.g. $(ab\$, b\$, \square, bb)$ which have to be considered.
2. The candidate set for the completion prefix system in figure 4.2 only has one more candidate which is $(db\$, b\$, \square, c)$ from rule r_{12} in combination with rule r_9. Note that rules r_2, r_3, and r_{11}, although their second component does overlap enough with the first of r_9, too, do not add new candidates. This is because their unconsumed completion is empty. For a different cause r_8 does not give a new candidate in combination with rule r_9: Here, the overlap is not big enough.

Remark 4.6. We wish to point out that $K_{R'} \cap R' = \textsc{Crules}(R')$.

Remark 4.7. At this point we see another reason for reducing the second components of the new rules with rules from $R' \setminus \$$. If we do not do this, the set of new rules that have to be added very often tends to be infinite, thus delivering infinite candidate sets. Properties of such infinite candidate sets would be difficult to prove. But even for finite maximal candidate sets a problem appears: Because of this reduction we are not able to make statements about all reductions possible with R'. We will overcome this restriction by the confluence property, which is conserved: Reducing the new rules in their second component corresponds to (mostly)[7] right canonical reductions in R'—and these exist because of the confluence.

Remark 4.8. As pointed out in section 4.3 we cannot assume that the original system is normalized. Because we are speaking about confluent rewriting systems we may assume that no left-hand side of a rule is a factor of another left-hand side. Furthermore, after using the prefix construction we again could drop rules which have a left-hand side reducible by another rule. In doing so we might drop rules with nonempty completions, and the information loss could be too high to check correctness. Therefore, this approach is not useful.

Now we want to distinguish which parts of a reduction are important for correctness checks and which are not. We will start from a reduction in R' and replace all reductions for which this is necessary by candidates.

Definition 4.12. *Let $K_{R'}$ be given. Let $w, w' \in \Gamma^*$, $s \in R'^+$ be a nonempty sequence of reduction rules such that for $w \in ¢ \cdot \Gamma^*_{Inner} \cdot \$$, $w' \in ¢ \cdot \Gamma^*_{Inner} \cdot \$ \cup \{Y\}$:*

$$w \underset{s}{\longrightarrow}^* w' \text{ and } |s| = l.$$

A candidate sequence for *s is a sequence $s' \in (R' \cup K_{R'})^+, s' = s'_1 s'_2 \cdots s'_n$ if there exists a partitioning $s_1, s_2, \ldots s_n$ with $1 \leq n \leq l$, $s_i \in R'^+$ and $s = s_1 s_2 \cdots s_n$ with*

$$w = w_0 \underset{s_1}{\longrightarrow}^* w_1 \underset{s_2}{\longrightarrow}^* w_2 \cdots \underset{s_n}{\longrightarrow}^* w_n$$

[7] We will give a formal definition for this "mostly" in definition 4.13.

which satisfies the following two conditions:

1. *For all* $1 \leq i \leq n$, $s'_i = (u, v, c_c, c_u)$ *and:*
   (a) *The two rules are identical, that is* $s_i = s'_i$ *and* (u, v) *is a rule of the original rewriting system* R *not working at the right-hand side of a word:*
   $s_i = s'_i$ *and* $(u, v) \in R \setminus \$$ *(and therefore* $c_c = c_u = \Box$*)*
   or
   (b) *The result of applying* s_i *is identical to that of applying* s'_i*, but they need not be equal (usually meaning* s'_i *is a shortcut for the reduction possible with* s_i*). Furthermore in that case we require* s'_i *to be candidate. More formally, this reads as:*
   $s_i \in R'^+$, $s'_i \in K_{R'}$, *and* $u \underset{s_i}{\longrightarrow}^* v$ *(then* $u \in \Gamma^+\$$ *and* $v \in \Gamma^* \cdot \$ \cup \{Y\}$ *holds).*
2. *For all* $1 \leq i < j \leq n$, $s'_i, s'_j \in K_{R'}$, $s'_i = (u, v, c_c, c_u)$, $s'_j = (u', v', c'_c, c'_u)$,
   and $s'_{i+1} \cdots s'_{j-1} \in (R \setminus \$)^*$*:*
   There exist u_1, u_2, v_2 *with* $u = u_1 u_2 \$$ *and* $v = u_1 v_2 \$$ *such that with appropriate* $w'_k, (i-1 \leq k \leq j-1)$*:*

$$
\begin{array}{ccc}
w_{i-1} & = & w'_{i-1} u_1 u_2 \$ = w'_{i-1} u \\
 & \underset{s_i}{\longrightarrow} & \\
w_i & = & w'_i u_1 v_2 \$ \\
 & \underset{s_{i+1} = s'_{i+1}}{\longrightarrow} & \\
w_{i+1} & = & w'_{i+1} u_1 v_2 \$ \\
 & \underset{s_{i+2} = s'_{i+2}}{\longrightarrow} & \\
w_{i+2} & = & w'_{i+2} u_1 v_2 \$ \\
 & \cdots & \\
 & \underset{s_{j-1} = s'_{j-1}}{\longrightarrow} & \\
w_{j-1} & = & w'_{j-1} u_1 v_2 \$ = w'_{j-1} v \\
 & \underset{s_j}{\longrightarrow} & \\
 & w_j &
\end{array}
$$

Also in this case we say s' is equivalent to s, *meaning that the intermediate reduction results of* s' *can be achieved with* s.

Remark 4.9. The above definition implies $v = u_1 v_2 \$ \in \mathrm{Suff}(u'\$)$ or $u'\$ \in \mathrm{Suff}(u_1 v_2 \$)$. Furthermore, if $w_j = Y$ then $s_j = s'_j$.

Note that a candidate sequence may have parts that are not candidates, but rules from the original rewriting system not valid at the right-hand end of a word. This has its origin in the following motivation for candidate sequences:

The partitioning of s and the corresponding candidate sequence s' are a means to detect which rule applications are "don't care" rules in the sense of proposition 4.1, part 1, and proposition 4.2. These "don't care" rules are the rules for which $s_i = s'_i \in R' \setminus \$$ in the definition. In contrast to this, if $s'_i \in K_{R'}$ they are more significant, because they reflect the part of the reduction which conveys completion information.

The following theorem shows that the definition of candidate sequences is sound:

Theorem 4.4. *Let* R' *be defined as in the above propositions and* $K_{R'}$ *be the corresponding candidate set. Then for every* $w \in \Sigma^*, ¢w\$ \xrightarrow[R']{}^* Y$ *there exists a sequence of rules of* R' *with* $w \xrightarrow[s]{}^* Y$ *such that there also exists a candidate sequence* s' *equivalent to* s.

Proof. Basically, we impose some restrictions on the reduction process which are possible because of the confluence of R'. To find s we use the algorithm in figure 4.4.

As one directly sees, the invariants of the two loops are equivalent to the conditions of candidate sequences. The variables s_i (for $i > 0$) directly provide the necessary partitioning.

The only critical part of the algorithm is the question of the existence of a candidate at the beginning of the while loop. By the construction of the candidate set, such a candidate exists if w_i is a reducible word: This is just the overlap condition used to find new candidates. □

```
reduce ¢w with any sequence s_0
   such that ¢w ---s_0--->* w_0 ∈ IRR(R' \ $); (*)
i := 0;
s := s_0;
s' := s_0;
while w_i ≠ Y do
    increment i;
    find a candidate s'_i = (u, v, c_c, c_u) ∈ K_R' and w_i with:
       w_{i-1}$ ---s'_i---> w_i$ or w_{i-1}$ ---s'_i---> Y = w_i;
    s' := s' · s'_i;
    set s_i ∈ R' · (R' \ $)* with u ---s_i--->* v
       to the sequence used in the computation of s'_i;
    s := s · s_i;
    if w_i ≠ Y then
      u_1 := longest common prefix of u and v;
      split w_i such that w_i = w'_i u_1 v_2;
      repeat
         search a rule r ∈ R' \ $ and w'_{i+1}
            such that w'_i u_1 ---r---> w'_{i+1} u_1; (**)
         if r not found then break; fi;    // leave repeat/until-loop
         increment i;
         s_i := r;
         s'_i := r;
         s := s · s_i;
         s' := s' · s'_i;
         w_i := w'_i u_1 v_2;
      until w'_i u_1 ∈ IRR(R' \ $);
    fi;
done;
```

Fig. 4.4: The algorithm which finds candidate sequences

Definition 4.13. *We call the reductions resulting from this algorithm* canonical candidate reductions. *If the reduction sequences generated before the while loop and by the repeat/until loop are right canonical—which could be enforced by adapting lines (*) and (**)—we call them* mostly rightmost canonical candidate reductions.

Sometimes we wish to extract the candidates from a candidate sequence:

Definition 4.14. *Let* $C' = \text{CPREF}(C)$ *be a completion prefix system with* $\text{RULES}(C) = R'$ *and* $K_{R'}$ *its candidate set. Then* $\text{CAND} : (R' \cup K_{R'})^* \to K_{R'}^*$ *is the morphism defined by its result on single elements of* $R' \cup K_{R'}$ *by:*

$$\text{CAND}(x) := \begin{cases} x & \text{if } x \in K_{R'} \\ \square & \text{otherwise.} \end{cases}$$

We can provide two sufficient conditions for the correctness of completion prefix systems:

Theorem 4.5. *Let* $C' = \text{CPREF}(C)$ *be a completion prefix system with* $\text{RULES}(C) = R'$ *and* $K_{R'}$ *its candidate set, both as in the last theorem. Assume for each* $w \in \Sigma^*, ¢w\$ \underset{R'}{\longrightarrow}^* Y$ *there exists a candidate sequence* s *which is valid in the completion prefix system* C'' *which has the set of completion rules* $R'' = R' \cup K_{R'}$. *Further we assume that the alphabet* Γ *of* C *has no true nonterminals, that is* $\Gamma = \Sigma \cup \{¢, \$, Y\}$.
Then C' *is correct.*

Proof. The condition implies the conditions of theorem 4.3 for each such w. □

Theorem 4.6. *Let* $C' = \text{CPREF}(C)$ *be a completion prefix system with* $\text{RULES}(C) = R'$ *and* $K_{R'}$ *its candidate set, both as in the last theorem. Assume for each* $w \in \Sigma^*, ¢w\$ \underset{R'}{\longrightarrow}^* Y$ *and each candidate sequence* s *with* $¢w\$ \underset{s}{\longrightarrow}^* Y$ *there exists a candidate sequence* s' *with* $\text{REW}(s) = \text{REW}(s')$ *and* s' *is valid in the completion prefix system* C'' *which has the set of*

completion rules $R'' = R' \cup K_{R'}$. *Further we assume that the alphabet* Γ *of* C *has no true nonterminals, that is* $\Gamma = \Sigma \cup \{¢, \$, Y\}$.
Then C' *is correct.*

Proof. The condition implies the conditions of theorem 4.5. □

Theorem 4.6 is insofar more restrictive as it does not only require the existence of a valid candidate sequence but also that for each candidate sequence there exists an equivalent one (w.r.t. the reduction process) which is valid.

4.8 Testing candidate sets

Before we are going to discuss methods to test candidate sets, we introduce some notations. These are helpful to abbreviate considerations about candidates and candidate sequences.

Definition 4.15. *Let* C *be a csCRLS,* $C' = \text{CPREF}(C)$, $R' = \text{RULES}(C')$, $K_{R'}$ *be the candidate set of* C', *and* $u = (u_1, u_2, u_3, u_4), v = (v_1, v_2, v_3, v_4) \in K_{R'}$.
We say u allows v in $K_{R'}$, $u \vdash_{K_{R'}} v$, *if* u_2 *and* v_1 *overlap such that their right-hand ends are matched together:*

$$u_2 \in \mathit{Suff}(v_1) \vee v_1 \in \mathit{Suff}(u_2).$$

Since this relation is only defined on the candidate set, v_1 and u_2 always end with a \$. This definition utilizes propositions 4.1 and 4.2 very similarly to the definition of candidate sequences (definition 4.12), since it only concentrates on those reductions possible at the right-hand end. The following propositions shows its relation to candidate sequences:

Proposition 4.4. *Let* C *be a csCRLS,* $C' = \text{CPREF}(C)$, $R' = \text{RULES}(C')$, *and* $R'' = R' \cup K_{R'}$ *with* $K_{R'}$ *being the candidate set of* C'.
Let $w \in ¢ \cdot \Gamma^*_{Inner} \cdot \$$, $w' \in ¢ \cdot \Gamma^*_{Inner} \cdot \$ \cup \{Y\}$ *and* $s \in R''^+$ *be a candidate sequence with* $w \underset{s}{\longrightarrow}^* w'$.

Assume $s = s_1 s_2 \cdots s_m$ *with* $m = |s|$ *and* $s_i \in R''$ *for* $(1 \leq i \leq m)$.
For $s' = \textsc{Cand}(s)$ *we address the candidates from* s *as following:*
$s' = s_{i_1} s_{i_2} \cdots s_{i_j} \cdots s_{i_n} = \textsc{Cand}(s)(1 \leq j \leq n)$ *with* $m \geq n \geq 2$ *such that* $i_j \in \{1, \ldots, m\}$ *and* i_j *is the index of the* j*-th candidate in* s *(of course, we assume* $i_j < i_{j+1}$*).*
Then for all $j < n$ $s_{i_j} \vdash_{K_{R'}} s_{i_{j+1}}$.

Proof. In order to speak in detail about the candidates, we name their components:

$$s_{i_j} = (u, v, c_c, c_u) \text{ and } s_{i_{j+1}} = (u', v', c'_c, c'_u),$$

where

$$u, v, u' \in \Gamma^* \cdot \$ \text{ and } v' \in \Gamma^* \$ \cup \{Y\}.$$

Let $w''v$ be the result after applying all rules up to s_{i_j}:

$$w \underset{s_1 \cdots s_{i_j - 1}}{\longrightarrow}{}^* w''u \underset{s_{i_j}}{\longrightarrow} w''v.$$

The rules used after that, but before $s_{i_{j+1}}$ are elements of $R \setminus \$$. Due to the definition of candidate sequences we know:
For v'', v''' with $v = v''v'''\$$ and $|v''| = \text{lcp}(u, v)$ all these intermediate rules may use v'' as right context, but they leave it unchanged. So there exists a w''' such that:

$$w''v \underset{s_{i_j+1} \cdots s_{i_{j+1} - 1}}{\longrightarrow}{}^* w'''v \underset{s_{i_{j+1}}}{\longrightarrow} w'.$$

Because $s_{i_j}, s_{i_{j+1}} \in K_{R'}$ they both work at the right-hand side of words. Furthermore, since C is a csCRLS and the construction of $\textsc{CPref}(C)$ does not introduce rules which add \$ to a word, this holds for all rules appearing in s too. Knowing that both v and u' are ending with a \$ implies that either $v \in \text{Suff}(u')$ or $u' \in \text{Suff}(v)$.

□

The following definition captures what is happening in valid rule (or candidate) sequences as defined in definition 4.8.

Definition 4.16. *Let C be a csCRLS, let $C' = \text{CPREF}(C)$, $K_{R'}$ be the candidate set of C', and $u = (u_1, u_2, u_3, u_4), v = (v_1, v_2, v_3, v_4) \in K_{R'}$.*

We say u allows *v* in *$K_{R'}$* with correct completion *if $u_4 \in$ Pref$(v_3 v_4)$ and $u \vdash_{K_{R'}} v$.*

That means the reduction represented by v may be applied safely after the one represented by u since the rest of the completion left by u fits to the completion of v.

Notation: $u \succ_{K_{R'}} v$.

Definition 4.17. *Let $C' = \text{CPREF}(C)$ be a completion prefix system, $R' = \text{RULES}(C')$, and $K_{R'}$ its candidate set.*

We define $K \subseteq K_{R'}$ is a valid set of *C' if for all $u = (u_1, u_2, u_3, u_4) \in K$ one of the following conditions holds:*

(i) $u_4 = \square$
(Fully consumed completion means that the next reduction(s) may be applied without regard of the completion; of course only until a new unconsumed completion appears.)

(ii) $u_4 = \$ \wedge u_2 = Y$
(After an accepting rule where only the word end marker is left as completion no further harm can be done. The reduction is finished...)

(iii) *for all $v = (v_1, v_2, v_3, v_4) \in K$ with $u \vdash_K v$*
exists $v' = (v_1, v_2, v'_3, v'_4) \in K$ with $u \succ_K v'$
(Whenever a reduction with v after u can take place, there is a variant v' of v with fitting next completion that does exactly the same w.r.t. the rewriting relation.)

Theorem 4.7. *Let $C' = \text{CPREF}(C)$ be a completion prefix system with $\text{RULES}(C') = R'$ and $K_{R'}$ its candidate set. Further we assume that the alphabet Γ of C has no true nonterminals, that is $\Gamma = \Sigma \cup \{¢, \$, Y\}$.*

If $K_{R'}$ is a valid set of *C', then C' is correct.*

Proof. For every word in $L_{C'}$ we can compute a candidate sequence s with theorem 4.4. For any two candidates in s, which all are in $K_{R'}$, that follow after each other we can exchange the second by one equivalent to it w.r.t. the rewriting relation such that it is fitting to the first w.r.t. the

completion. Starting with the second candidate in s and repeating this to the last we get a candidate sequence s' which satisfies the conditions of theorem 4.5. □

Remark 4.10. We used the weaker theorem 4.5. But since the exchange process in the proof works for any candidate sequence, also the conditions of theorem 4.6 are satisfied.

Theorem 4.8. *Let $C' = \text{CPREF}(C)$ be a completion prefix system with $\text{RULES}(C') = R'$, $K^f_{R'}$ its candidate set which is finite.*
Then it is decidable if $K^f_{R'}$ is a valid set.

Proof. Because $K^f_{R'}$ is finite an algorithm can check conditions (i) and (ii) for each single candidate $u \in K^f_{R'}$ and condition (iii) for each pair $u, v \in K^f_{R'}$ with $u \vdash_{K^f_{R'}} v$. □

Example 4.6. Let $\Sigma = \{a, b, c, d\}$ and $\Gamma = \Sigma \cup \{¢, \$, Y\}$ with the usual meaning of $¢, \$$, and Y. Now we define three csCRLSs C_1, C_2, C_3 by supplying rewriting systems R_1, R_2, R_3:

$$\begin{aligned}
R_1 &= \{(bcc, bc), (abd, add), (¢add\$, Y)\} \\
R_2 &= \{(bcd, bd), (abd, add), (¢add\$, Y)\} \\
R_3 &= \{(bcc, bc), (bcd, bd), (abd, add), (¢add\$, Y)\} = R_1 \cup R_2
\end{aligned}$$

The languages of these systems are:

$$\begin{aligned}
L_{C_1} &= \{abd, add\} \\
L_{C_2} &= \{abd, add, abcd\} \\
L_{C_3} &= \{add\} \cup \{ab\} \cdot \{c\}^* \cdot \{d\}
\end{aligned}$$

The completion rules of the corresponding completion prefix sets C'_1, C'_2, and C'_3 are:

$$\begin{aligned}
R'_1 = \{&(bcc, bc, \square, \square), \quad (bc\$, b\$, \square, c), \quad (abd, add, \square, \square), \quad (ab\$, ad\$, \square, d), \\
&(¢add\$, Y, \square, \$), (¢ad\$, Y, d, \$), (¢a\$, Y, dd, \$), (¢\$, Y, add, \$)\} \\
R'_2 = \{&(bcd, bd, \square, \square), \quad (bc\$, b\$, \square, d), \quad (abd, add, \square, \square), \quad (ab\$, ad\$, \square, d), \\
&(¢add\$, Y, \square, \$), (¢ad\$, Y, d, \$), (¢a\$, Y, dd, \$), (¢\$, Y, add, \$)\}
\end{aligned}$$

$$R'_3 = \{(bcc, bc, \square, \square), \quad (bc\$, b\$, \square, c), \quad (bcd, bd, \square, \square), \quad (bc\$, b\$, \square, d),$$
$$(abd, add, \square, \square), (ab\$, ad\$, \square, d), (¢add\$, Y, \square, \$),$$
$$(¢ad\$, Y, d, \$), (¢a\$, Y, dd, \$), (¢\$, Y, add, \$)\} = R'_1 \cup R'_2$$

For $i \in \{1, 2, 3\}$ the candidate sets are $K_{R'_i} = \textsc{Crules}(R'_i)$. Now we look at the word abc which is not a prefix of a word in L_{C_1} but a prefix of a word in L_{C_2} and L_{C_3}. Inspecting the completion prefix systems shows:

C_1: $abc \in L_{C'_1}$. So, C'_1 cannot be correct. We see that

$$(bc\$, b\$, \square, c) \vdash_{K_{R'_1}} (ab\$, ad\$, \square, d)$$

but there is no candidate $v' = (ab\$, ad\$, v_3, v_4)$ with $v' \in K_{R'_1}$ and $(bc\$, b\$, \square, c) \succ_{K_{R'_1}} v'$. So $K_{R'_1}$ is not valid.

C_2: Checking $K'_{R'_2}$ shows that it is valid. With theorem 4.7 C'_2 is correct.

C_3: A close inspection shows that C'_3 is correct, because instead of $(bc\$, b\$, \square, c)$ one can always use $(bc\$, b\$, \square, d)$. But $K_{R'_3}$ is not valid, the argument is the same as for C_1. We see that we cannot use theorem 4.7 to show that is is correct.

A first approach derived from C_3 could be to exchange the first, not the second, candidate by an equivalent one in the definition of valid sets. But this would only cause a symmetric problem. In consequence we have to find a less restrictive condition for correctness. This is the aim of the next section.

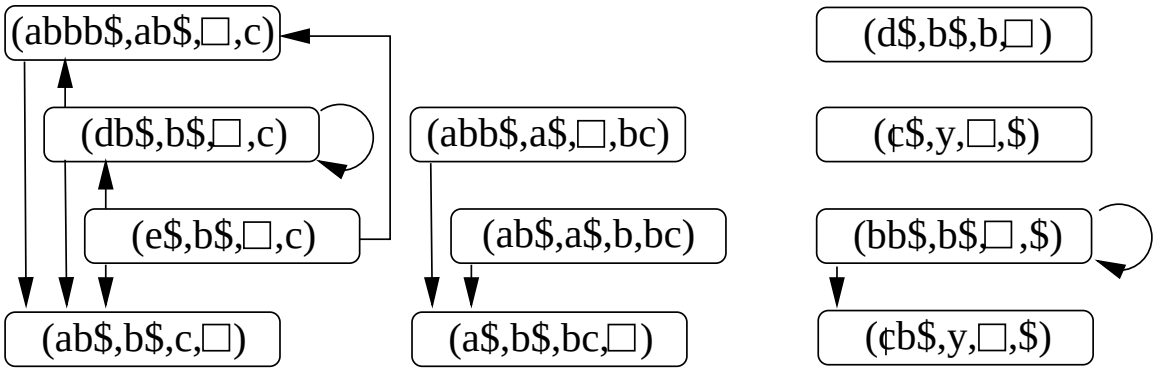

Fig. 4.5: An example for a candidate graph

4.9 A weaker condition for correctness

Expecting a candidate set to be valid is far to strict a condition. We recall that with theorem 4.6 we do not assume that all candidates are expected to be valid. That means, there may be candidates which are not part of any correct reduction of a prefix. This does not automatically lead to an incorrect prefix system.

We now introduce two graph notations which allow to discuss properties of the reflexive, transitive closures of the relations $\vdash_{K_{R'}}$ and $\succ_{K_{R'}}$. The first definition, candidate graphs, reflects reductions with correct completions.

Definition 4.18. *Let $C' = \text{CPREF}(C)$ be a completion prefix system with $\text{RULES}(C') = R'$ and $K_{R'}$ its candidate set.*

The candidate graph of $K_{R'}$ *is the directed graph*

$$\begin{aligned} G^C_{K_{R'}} := (K_{R'}, \{(u,v) \mid & \\ & u, v \in K_{R'} \wedge u \succ_{K_{R'}} v \\ & \wedge \exists u_1, u_2, u_3 \in \Gamma^*, u_4 \in \Gamma^+ \colon u = (u_1, u_2, u_3, u_4)\}). \end{aligned}$$

We also allow the shorter notation $G^C_{R'}$.

Candidate graphs are a way to describe $\text{CAND}(s)$ for all basic valid reductions s: Each path in the candidate graph ending with a vertex that has out-degree zero describes such a reduction.

Example 4.7. Figure 4.5 depicts the candidate graph for the candidate set of the completion prefix system given in figure 4.2.

On the other hand, we want to know which reductions are possible if the completion information is not regarded. This leads to the following definition:

Definition 4.19. *Let $C' = \text{CPREF}(C)$ be a completion prefix system with $\text{RULES}(C') = R'$ and $K_{R'}$ its candidate set.*

The rewriting graph of $K_{R'}$ *is the directed graph*

$$
\begin{aligned}
G^R_{K_{R'}} := (\ & \textsc{Rew}(K_{R'}),\\
& \{(\textsc{Rew}(u), \textsc{Rew}(v)) |\\
& \quad u, v \in K_{R'} \wedge u \vdash_{R''} v\\
& \quad \wedge \exists u_1, u_2, u_3 \in \Gamma^*, u_4 \in \Gamma^+ : u = (u_1, u_2, u_3, u_4)\}).
\end{aligned}
$$

Again, we may also use $G^R_{R'}$.

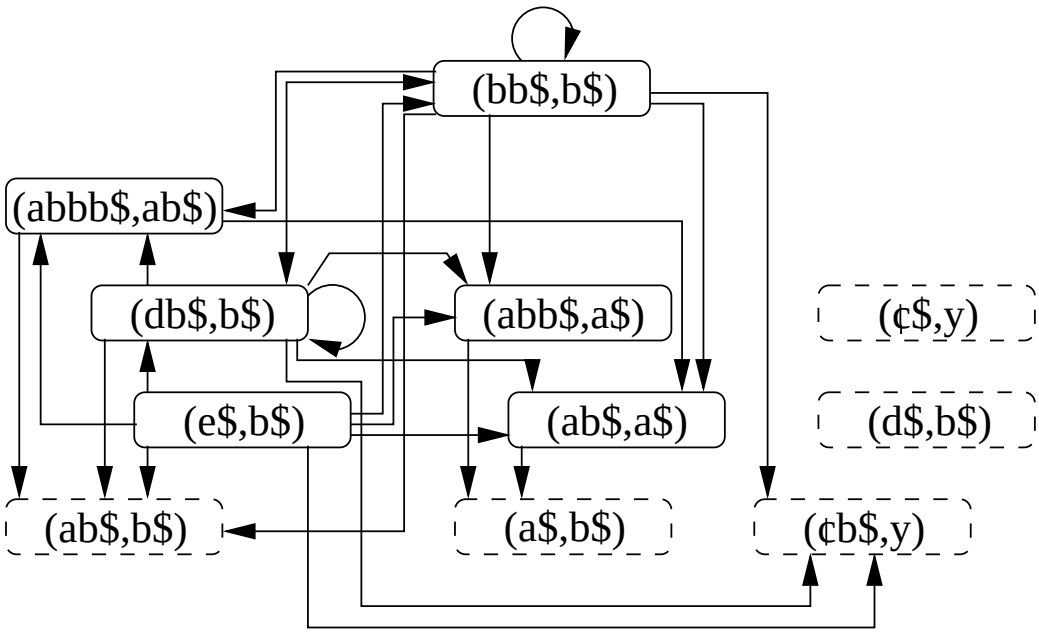

Fig. 4.6: An example for a rewriting graph

Example 4.8. Figure 4.6 depicts the rewriting graph for the candidate set of the completion prefix system given in figure 4.2. For clarity, vertices which are derived from a completion rule with empty unconsumed completion have a dashed border.

Theorem 4.9. *Let* $C' = \textsc{CPref}(C)$ *be a completion prefix system with* $\textsc{Rules}(C') = R'$, $K_{R'}$ *its candidate set, and* $R'' = R' \cup K_{R'}$. *Let* $w, w' \in \Gamma^*$ *and* $s \in R''^+$ *be a candidate sequence such that for* $w \in ¢ \cdot \Gamma^*_{Inner} \cdot \$$,

$w' \in ¢ \cdot \Gamma^*_{Inner} \cdot \$ \cup \{Y\}$ and $w \underset{s}{\longrightarrow}^* w'$. Let $s' = \textsc{Cand}(s)$, and $G^R_{K_{R'}}$ be the rewriting graph of $K_{R'}$.
Then there is a partitioning of s' such that $s' = s_1 s_2 \cdot s_i \cdots s_n$, $s_i \in K^+_{R'}$ for all $1 \leq i \leq n$ and $\textsc{Rew}(s_i)$ is a path in $G^R_{K_{R'}}$.

Proof. The partitioning can be accomplished by breaking s' up after each candidate with an empty unconsumed completion. Then it is clear from the definition of the rewriting graph that it must contain the corresponding paths. □

Definition 4.20. *We call the partitioning acquired by the last theorem* path partitioning of s'. *For a part s_i as above we call $\textsc{Rew}(s_i)$ the* rewriting path of s_i.

Theorem 4.10. *Under the same conditions as theorem 4.9 we further assume that s is a valid reduction with finished completion. Then each s_i of the partitioning of that theorem is a basic valid reduction. Furthermore, each s_i is a path in the corresponding candidate graph $G^C_{R'}$.*

Proof. That s_i is valid and also basic follows from the definition of (basic) valid reductions and from the fact that in the proof of theorem 4.9 we always split when a completion is fully consumed. Since each s_i is valid and basic and because the graph $G^C_{R'}$ reflects the conditions of basic valid reductions s_i must be a path in the graph. □

Remark 4.11. It is important that s has a finished completion. Otherwise, s_n would not have a finished completion and therefore, given the definition of basic valid reductions, s_n could not be a basic valid reduction. Furthermore, one should notice that the graph will be infinite if the candidate set is infinite.

Given these results we can speak about correctness of prefix systems by inspecting the two types of graphs we defined:

Theorem 4.11. *Assume the conditions of theorem 4.9 and that the terminal alphabet of C is $\Sigma = \Gamma_{inner}$ (C has no true nonterminals). Assume for each path s in $G^R_{K_{R'}}$ there is a path s' in $G^C_{K_{R'}}$ with $\textsc{Rew}(s') = s$. Then C' is correct.*

Proof. This formulates the conditions of theorem 4.6 in terms of graphs. □

Utilizing the fact that REW is used as morphism on sequences of candidates, we can also work with its inverse. Then we can use the following notation for the conditions of this theorem:

$$\forall s \in \textsc{Path}(G^R_{K_{R'}}) : \textsc{Rew}^{-1}(s) \cap \textsc{Path}(G^C_{K_{R'}}) \neq \emptyset,$$

where $\textsc{Path}(G^R_{K_{R'}})$ is a shorthand for the set of paths in $G^R_{K_{R'}}$.

Example 4.9. We now continue example 4.6. In figure 4.7 the candidate and rewriting graphs are shown. Note that all three rewriting graphs are identical. We see that all graphs are cycle-free. So it is easily possible to verify the conditions of theorem 4.11. Now we are able to show that C'_3 is correct, and we see that for C'_2 we can show correctness with this method, too.

Unfortunately we are not able to look at all possible paths, since the graphs may contain cycles. But the next theorem shows that this problem can be circumvented:

Theorem 4.12. *Let G^R_K be the rewriting graph of a finite candidate set K and G^C_K be the candidate graph of the same set. Then it is decidable if for each path s in G^R_K there is a path s' in G^C_K with* $\textsc{Rew}(s') = s$.

Proof. Let L be the language of all paths in G^C_K. Since K is finite, this language is regular. REW is a (homo)morphism from the set of all candidate sequences to the set of all (rewriting) rule sequences. Therefore $\textsc{Rew}(L)$ is a regular language too, and

$$s \in \textsc{Rew}(L) \Longleftrightarrow \exists s' \in L : \textsc{Rew}(s') = s. \quad (*)$$

On the other hand, let L' be the language of all paths in G^R_K. Then $\textsc{Rew}(L) \subseteq L'$ holds. With $(*)$ we know:

$$s \in L' \setminus \textsc{Rew}(L) \Longrightarrow \nexists s' \in L : \textsc{Rew}(s') = s.$$

So for each path s in the rewriting graph there exists a path s' in the candidate graph with $\textsc{Rew}(s') = s$ if and only if $\textsc{Rew}(L) = L'$, which is decidable. $\square$

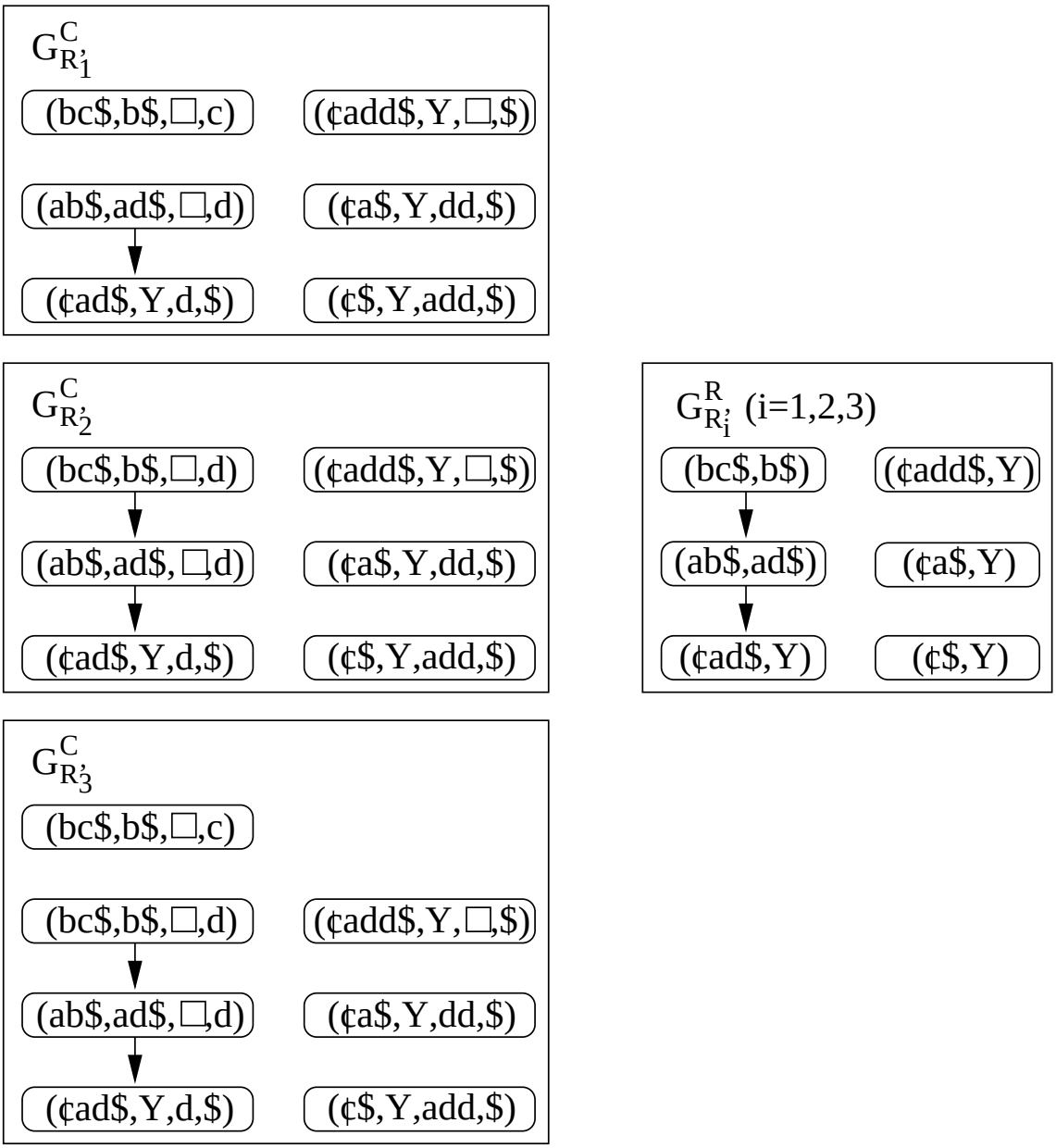

Fig. 4.7: Candidate and rewriting graphs for example 4.6

An approximate upper bound for the time complexity can be given, too. We only need to decide if $L' \subseteq \textsc{Rew}(L)$, which is equivalent to $L' \setminus \textsc{Rew}(L) = L' \cap \overline{\textsc{Rew}(L)} = \emptyset$, where $\overline{\textsc{Rew}(L)}$ is the complement of $\textsc{Rew}(L)$.

Let n be the number of candidates (the size of K). Clearly, this is an upper bound for the number of states necessary for a finite automaton accepting L[8]. This is also an upper bound for an automaton A accepting $\textsc{Rew}(L)$, and for an automaton A' for L', too. Unfortunately, A' directly constructed from the graphs may be nondeterministic. So an Automaton A'' for $\overline{\textsc{Rew}(L)}$ could have $O(2^n)$ states in the worst case.

Therefore, an automaton for $L' \cap \overline{\textsc{Rew}(L)}$ could have $O(n \cdot 2^n)$ states in the worst case. With a standard algorithm for computing the reflexive, transitive closure of the state transitions we can decide, whether this automaton accepts the empty language. So we get $O((n \cdot 2^n)^3)$ as upper bound for the time necessary to decide our problem.

This estimation probably is no good upper bound. The argument does not take advantage of the fact that the automata for $\textsc{Rew}(L)$ and L' are not arbitrary automata. Especially their states are closely linked to each other by $\textsc{Rew}$. Therefore decidability in polynomial time cannot be excluded for sure.

4.10 True terminal correctness

So far we considered the case where only the special symbols ¢, \$, and Y are true nonterminals. This section will cover the problem $\Gamma \setminus \Sigma \supset \{¢, \$, Y\}$, that is if there are more nonterminals. We will discuss two very simple sufficient conditions for correctness in this case. Since even these simple conditions are not very promising (in terms of complexity and decidability) one has to find alternative strategies. This will be discussed in the next chapter, which is about more practical applications.

Example 4.10. Assume $\Sigma = \{a, b, c\}$ and $\Gamma = \Sigma \cup \{A, B, C, ¢, \$, Y\}$. Now look at the following rule set:

[8] Off course, there exist automata with more states accepting the same language.

$$R = \{(ABC, \square), (BBC, BC), (a, A), (b, B), (c, C), (¢\$, Y)\}.$$

Then the rule set R' of the corresponding completion prefix system is:

$$\begin{aligned} R' = \text{CPREF}(R) = \{ & (ABC, \square, \square, \square), (AB\$, \$, C, \square), (A\$, \$, BC, \square), \\ & (BBC, BC, \square, \square), (BB\$, B\$, \square, C), \\ & (a, A, \square, \square), (b, B, \square, \square), (c, C, \square, \square), \\ & (¢\$, Y, \square, Y)\}. \end{aligned}$$

In figure 4.8 the candidate and rewriting graphs can be found. Observe that if we interprete REW as graph morphism it is an isomorphism between the two graphs. In consequence, with the results of the last section we know:

$$\forall w \in \Sigma^*, ¢w\$ \xrightarrow[R']{}^* Y : \exists w' \in \{a, b, c, A, B, C\}^* : ¢ww'\$ \xrightarrow[R]{}^* Y.$$

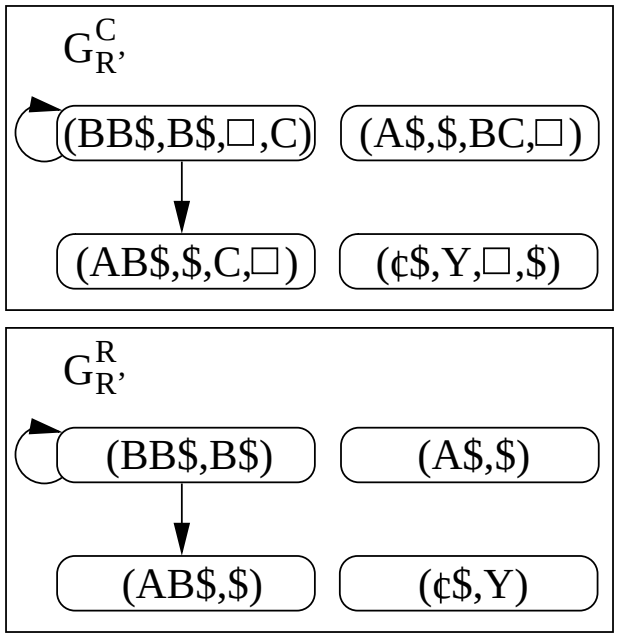

Fig. 4.8: Candidate and rewriting graphs for example 4.10

How can we show that there is a completion only consisting of terminals for all words accepted by R'?

$$\textbf{Question: } \forall w \in \Sigma^*, ¢w\$ \underset{R'}{\longrightarrow}^* Y : \exists w' \in \Sigma^* : ¢ww'\$ \underset{R}{\longrightarrow}^* Y?$$

In this example, this is very simple, because of the three rules that rewrite the terminals with their nonterminal counterpart.

Example 4.11. We consider a (part of a) different completion system, again called R', with the same alphabets, having the following completion rules (amongst others):

$$(AB\$, C\$, B, C), (C\$, \$, C, \square), \text{ and } (¢\$, Y, \square, \$).$$

Now assume this system has no rule directly rewriting a terminal (or a terminal word) to B, but the following rules:

$$(Ab, AB, \square, \square), (a, A, \square, \square), (ABbC, ABBC, \square, \square), \text{ and } (c, C, \square, \square).$$

We know:

$$¢ab\$ \underset{R'}{\longrightarrow}^* ¢AB\$ \underset{R'}{\longrightarrow}^* Y.$$

Note that we need not consider the first reduction in R' further: The input which is checked by a completion prefix system must consist of terminals anyhow; it is of no interest if a word not reachable from a terminal word might be reducible to Y.

Furthermore, if R again is the original system (one can derive the original rules from the completion rules):

$$¢abBC\$ \underset{R}{\longrightarrow}^* ¢ABBC\$ \underset{R}{\longrightarrow}^* Y.$$

So $w' = BC$ is the nonterminal completion provided by R'. We also know:

$$¢abbc\$ \underset{R}{\longrightarrow}^* ¢ABBC\$ \underset{R}{\longrightarrow}^* Y.$$

But we know even more about the system R:

$$ABbc \underset{R}{\longrightarrow}^* ABBC.$$

Because in the candidate graph the completion rules of this example do not lead to a cycle, there is only the single completion BC. To be more general, the set of possible completions generated by these rules is finite. The following example shows that this is not always the case:

Example 4.12. Assume a completion prefix system R' had the following rule:

$$(AA\$, A\$, B, B).$$

What if for all $n \geq 2$ there is a word $w \in \Sigma^*$ such that

$$¢w\$ \underset{R'}{\longrightarrow}^* ¢A^n\$ \underset{R'}{\longrightarrow}^* Y?$$

Then we would have to check infinitely many completions. This might be possible for this example, because this set is regular, but one cannot guarantee this. The nonterminal completions could be anything describable by the original rewriting system.

There is a crucial difference to example 4.10. The rule $(BB\$, B\$, \square, C)$ in this earlier example does not lengthen the completion, in contrast to the rule given just above.

Now we are going to gather the observations of these examples into a series of lemmas.

Lemma 4.2. *Let C be a csCRLS and $C' = \text{CPREF}(C)$, with alphabets Σ and Γ where $\Gamma \setminus \Sigma \supset \{¢, \$, Y\}$. Assume for the csCRLS C'' which is the same as C except for its terminal alphabet Σ' being $\Sigma' = \Gamma \setminus \{¢, \$, Y\}$ we know $\text{CPREF}(C'')$ is correct.*

If for every basic valid reduction in C' of the form

$$¢w\$ \underset{C'}{\longrightarrow}^* w'(w' \in (\{¢\} \cdot \Gamma^* \cdot \{\$\}) \cup \{Y\})$$

having a consumed completion c and an empty unconsumed completion or an unconsumed completion \$ such that

$$¢wc\$ \xrightarrow[C]{}^* w'$$

the following holds:

$$\exists c' \in \Sigma^* : ¢wc' \xrightarrow[C]{}^* ¢wc \textit{ if the unconsumed completion is empty}$$

or

$$\exists c' \in \Sigma^* : ¢wc'\$ \xrightarrow[C]{}^* ¢wc\$ \textit{ if the unconsumed completion is } \$,$$

then C' is correct.

Proof. Assume $w \in (\Gamma \setminus \{¢, \$, Y\})^*$ and $¢w\$ \xrightarrow[C']{}^* Y$. With the assumptions we know there is a $c \in (\Gamma \setminus \{¢, \$, Y\})^*$ such that

$$¢wc\$ \xrightarrow[C]{}^* Y$$

and a valid reduction $s \in C'^*$ such that

$$¢w\$ \xrightarrow[s]{}^* Y.$$

Let $n \geq 1$ be the number of basic valid reductions in s. We make an induction over n:

Induction basis. $(n = 1)$ s provides a consumed completion c and the unconsumed completion \$. So we know

$$¢wc\$ \xrightarrow[C]{}^* Y$$

and with the assumptions there is a $c' \in \Sigma^*$ with

$$¢wc'\$ \xrightarrow[C]{}^* ¢wc\$ \xrightarrow[C]{}^* Y.$$

So the claim holds with the terminal completion c'.

Induction claim. Assume for $n > 1$ the properties to be proved hold.

Induction step. $(n \to n+1)$ Let $s = s_1 s_2$ where s_1 is the first basic valid reduction appearing in s. Then s_1 provides a consumed completion c and an unconsumed completion $c_u \in \{\Box, \$\}$. We only show the case of $c_u \neq \$$, since if only a \$ is left as completion this is fixed and no more other nonterminals can be introduced, but in principle this case is analogous.

If $c_u \neq \$$ there is a rest of the completion $u \in \Gamma \setminus \{¢, \$, Y\}$ providing:

$$¢w\$ \underset{s_1}{\longrightarrow}^* ¢w'\$ \underset{s_2}{\longrightarrow}^* Y$$

for suitable w' and there is a c':

$$¢wc' \underset{C}{\longrightarrow}^* ¢wc \quad (*)$$

Therefore,

$$¢wc'u\$ \underset{C}{\longrightarrow}^* ¢wcu\$ \underset{C}{\longrightarrow}^* ¢w'u\$ \underset{C}{\longrightarrow}^* Y.$$

With the induction claim and $(*)$ there is a $u' \in \Sigma^*$ such that

$$¢wc'u'\$ \underset{C}{\longrightarrow}^* ¢wcu'\$ \underset{C}{\longrightarrow}^* ¢w'u'\$ \underset{C}{\longrightarrow}^* Y.$$

As mentioned above, the case c_u is very similar. In that case u and u' in the last two steps are the empty word. $\Box$

Candidate graphs describe basic valid reductions, too. We are going to use this for testing terminal correctness.

Definition 4.21. *Let K be a candidate set and $s = s_1 \cdots s_i \cdots s_n$ with $s_i \in K$ $(1 \leq i \leq n)$, and $s_i = (u_i, v_i, c_i, w_i)$. Assume $v_n = Y$ or $w_i = \Box$. Then $c_1 \cdots c_n$ is the completion of s. The set of all completions for all such s is called the completion set of K, for short* $\textsc{Comp}(K)$.

Lemma 4.3. *Assume K is a finite candidate set. Then* $\textsc{Comp}(K)$ *is finite if and only if the candidate graph G_K^C has no cycles that provide nonempty completions. That means, all cycles are of the form:*

$$(u_1, v_1, \Box, c_1) \cdots (u_i, v_i, \Box, c_i) \cdots (u_n, v_n, \Box, c_n)(u_1, v_1, \Box, c_1)$$

with $(1 \leq i \leq n)$, $n > 1$, $c_i \in \mathit{Pref}(c_{i+1})$ *for* $i < n$, *and* $c_n \in \mathit{Pref}(c_1)$.

Proof. Note that the condition linking the unconsumed completions holds for all cycles because of the definition of the edge set of candidate graphs. Furthermore, if no consumed completion appears in the cycle then $c_i = c_1$ for all $1 \leq i \leq n$.

$\Longrightarrow$: (by contraposition) Assume there is a cycle containing a nonempty consumed completion. Then we can provide arbitrarily long completions by simply looping through that cycle arbitrarily often.

$\Longleftarrow$: Since no cycle changes the completion, and no cycle adds an unconsumed completion, $\textsc{Comp}(K)$ can be generated by the set of cycle-free paths, and must be finite in consequence. $\Box$

Lemma 4.4. *Let K be a finite candidate set. Then it is decidable if* $\textsc{Comp}(K)$ *is finite.*

Proof. If G_K^C has a cycle providing nonempty completions, as in the last lemma, then it has a simple cycle with that same property: Given an arbitrary cycle that is not simple and having a nonempty consumed completion, one can look at all its simple cycles. Either they fulfill the property, or they have an empty consumed completion. In the latter case, one removes that simple cycle and repeats looking for other simple cycles. Therefore, the graph has a cycle with nonempty consumed completion if and only if it has a simple cycle with the same property.

The set of all simple cycles of a finite graph is finite, therefore the problem is decidable. $\Box$

Theorem 4.13. *Let C be a csCRLS and $C' = \textsc{CPref}(C)$. Let K be the finite candidate set of C' with* $\textsc{Comp}(K)$ *being finite, too. Let $k \geq 0$ be a fixed number. Then for all basic valid reductions s it is decidable if their completion can be reduced from a terminal word in k reduction steps under the following restrictions:*

(i) As left context, only the left-hand side of the rewriting part of the first candidate in the reduction may be used.

(ii) As right context only the symbol \$ may be used if it is the unconsumed completion of the last candidate. Otherwise, no right context is allowed.

More formally this means the following. Assume the first candidate in s is (u_1, v_1, c_1, w_1), the last is (u_n, v_n, c_n, w_n) (with $w_n \in \{\Box, \$\}$). Assume the completion provided by s is c. It is decidable for all such s if there exists a $c' \in \Sigma^*$:

$$u_1 c' w_n \underset{C}{\longrightarrow}^{\leq k} u_1 c w_n.$$

Proof. As mentioned above, all candidate sequences of basic valid reductions correspond to a path in the candidate graph. If $\textsc{Comp}(K)$ is finite, we only need to look at cycle-free paths. Let s' be such a path, representing a basic valid reduction. We may assume $s' = s_1 \cdots s_i \cdots s_n$ with $s_i \in K$ $(1 \leq i \leq n)$, and $s_i = (u_i, v_i, c_i, w_i)$. Assume $v_n = Y$ or $w_i = \Box$. Then $c = c_1 \cdots c_n$ is the completion as above. The number of $c' \in \Gamma^*$ with $u_1 c' w_n \underset{C}{\longrightarrow}^{\leq k} u_1 c w_n$ is finite, and so we only have to compute all of them and check if one only consists of terminals. $\Box$

4.11 A note on (un)decidability

Obviously, in the last theorem k is providing a further finiteness condition. But for arbitrary k one can find a prefix system which is correct and has a completion that cannot be derived from a terminal word in k steps. This is very simple. Just take the alphabets $\Sigma = \{a\}$ and $\Gamma = \{a, A, ¢, \$, Y\}$ and the following two rules:

$$(a, A) \text{ and } (¢A^{k+1}\$, Y).$$

The prefix system given by such two rules always is correct, but for the completion prefix rule $(¢, \$, A^{k+1}, \$)$ one needs $k+1$ steps to reduce a terminal word to the completion A^{k+1} of the prefix rule.

Unfortunately, this does not prove undecidability of correctness, it only shows that even under such strong restrictions as that of the finiteness of $\textsc{Comp}(K)$ deciding correctness can be very difficult.

One possible strategy for proving undecidability would work with simulations of Turing machines, similar to [OKK97]. If it is possible to show that the set of CRLS that have a confluent and weight-reducing prefix system are a basis of the recursively enumerable languages (an interesting question on its own account), one could use the following argument:

Definition 4.22. *Let $u, v \in \Sigma^*$. Then the* shuffle product of u and v, *denoted with $u \sqcup\!\sqcup v$ is defined as:*

$$u \sqcup\!\sqcup v := \{u_1 v_1 u_2 v_2 \cdots u_i v_i \cdots u_n v_n \mid 1 \leq i \leq n, u_i, v_i \in \Sigma^*, \\ u = u_1 u_2 \cdots u_n, v = v_1 v_2 \cdots v_n\}.$$

For $L_1, L_2 \subseteq \Sigma^*$, the shuffle product is defined as:

$$L_1 \sqcup\!\sqcup L_2 := \{u \sqcup\!\sqcup v \mid u \in L_1, v \in L_2\}.$$

Assume for arbitrary recursively enumerable languages $L \subseteq \Sigma^*$ there is a csCRLS C such that for every word $w \in L$ there is a k_0 such that $(\{w\} \sqcup\!\sqcup \{\sharp\}^*) \cdot \sharp^k \in L_C$ for every $k \geq k_0$, where $\sharp$ is a new symbol and $\sqcup\!\sqcup$ is the shuffle product. We call L_C the *shuffle basis of* L.

Assume that this C can be constructed from a deterministic Turing machine M accepting L. Without going into the details (we refer the reader to [OKK97]), for every w the corresponding k can be computed from the number of steps that M needs to accept w. Assume further that $\text{CPREF}(C)$ is defined for such a C and for every L.

For every recursively enumerable language $L \subseteq \Sigma^*$ there is a Turing machine M which accepts Σ^* if L is not empty and $\emptyset$ otherwise.[9] Clearly, given M, it is not decidable which of the two holds. With the assumptions we can build C and $C' = \text{CPREF}(C)$ from M.

Assume it is decidable whether C' is correct. We know that C' always accepts the empty word. So, C' is not correct, if $L = \emptyset$, since in that case $L_C = \emptyset$. On the contrary, assume $L = \Sigma^*$. Then $\text{Pref}(L_C) = (\Sigma \cup \{\sharp\})^* \subseteq$

[9] Ignoring the input (that is, deleting it before it starts the real computation), M simply dove-tails over all words in Σ^* and computation steps of a Turing machine M' accepting L. As soon as a word is accepted by M', M accepts, too.

$L_{C'}$. In consequence, $L_{C'} = (\Sigma \cup \{\sharp\})^*$, because there are no other words which could be accepted by C'. So C' is correct. Combining that, we get:

(i) $L = \emptyset \Longrightarrow C'$ is not correct.
(ii) $L = \Sigma^* \Longrightarrow C'$ is correct.

Which means $L = \Sigma^*$ if and only if C' is correct. So, under the given assumptions, decidability of correctness leads to decidability of the emptiness problem for arbitrary recursively enumerable languages, leading to a contradiction.

Theorem 4.14. *Correctness of prefix systems is undecidable if it is possible to construct a csCRLS C for every recursively enumerable language L such that L_C is the shuffle basis of L and* $\text{CPREF}(C)$ *is defined.*

Proof. (given above) □

Note that the reverse need not be true.

Remark 4.12. The shuffling is *not* the problem for such a hypothetic construction. It rather might be that providing confluence and weight-reduction of the prefix system cannot be achieved. Or, which would be even worse, it is possible that no *construction* exists, but a system for a shuffle basis with confluent and weight-reducing prefix system exists for every recursively enumerable language, anyhow.

As one can see, the issue of decidability of the correctness of prefix systems is closely linked to the fact that CRL are a basis of the recursively enumerable languages. We conjecture that correctness is not decidable in general. Therefore, in our opinion, the more pragmatic approach of testing properties which ensure correctness is justified.

Chapter 5

Applications

This chapter deals with two practical aspects of CRL. In the first section we consider the relation to the well known deterministic context-free languages w.r.t. prefix systems. The second section describes an experimental development environment which can be used to edit and test CRLSs.

5.1 Deterministic context-free languages

Shift/reduce-parsers

General deterministic pushdown automata, as they can be found in textbooks like [Har78], which are the original means to define deterministic context-free languages [GG65] , are not used for practical applications. They are relatively difficult to handle. Tools like YACC use shift/reduce-parsers instead. These are generated from special context-free grammars. In the case of YACC, LALR(1)-grammars are used. Generally, shift/reduce-parsers can be built from LR(k)-grammars [Knu65]. Three qualities of these parsers are important:

1. The LR(k)-grammars and shift/reduce-parsers characterize the deterministic context-free languages.
2. Properties of the grammars are reflected. That means, having a grammar with LR(k)-property and the shift/reduce- parser generated for it, one can use the computation of the parser to gain a parse for an accepted input word in that same grammar.
3. Using a lookahead of k symbols allows more flexibility. Especially, the bigger the lookahead, the bigger the class of grammars having the LR(k)-property. This reflects the fact that with the lookahead the parser handles unambiguity during the parsing process (while

the grammars themselves always are unambiguous). Since this does not change the class of accepted languages, one has a more general class of automata for deterministic context-free languages.

We do not want to examine the details of LR(k)-grammars. Therefore, the construction of the shift/reduce-parsers will not be discussed in the following definition. For details, we refer the reader to [Knu65] and [Joh75] or textbooks [App98,AU72,AU73,Har78,HU79,SSS90]. Furthermore, the definition is different from standard textbooks in some minor details, but this is only a matter of representation, not of the computational power. We choose this representation in order to make the translation to csCRLSs more easily describable.

Before we come to the definition of shift/reduce-parsers it should be noted that for convenience we only discuss the case $k \geq 1$. LR(0)-grammars have to be handled slightly different in some cases, because they are not equally powerful as LR(k)-grammars for $k \geq 1$. This will not pose a restriction on our results, since any LR(0)-grammar is an LR(1)-grammar, too.

Definition 5.1. *Let $G = (\Sigma, N, S, P)$ be a an LR(k)-grammar ($k \geq 1$), where S does not appear on the right-hand side of a rule in P. A* shift/reduce-parser *M_G for G with lookahead k is a deterministic device, defined by a 7-tuple $M_G = (\Sigma, \Gamma, s_0, K, Q, \$, T)$ where the elements of the tuple have the following meaning:*

Σ is the input alphabet*,*
Γ is the table set*,*
$s_0 \in \Gamma$ is the initial table*,*
$K = \{(s_0, \Box)\} \cup \Gamma \times (\Sigma \cup N)$ is the stack alphabet*,*
$Q = K \cup \{accept, error\}$ is the state set*, $Q \setminus \Gamma = \{accept, error\}$,*
accept is the accepting state*,*
error is the error state*,*
$\$ \notin \Sigma \cup N$ is the end marker*, and*
T is the transition table*. The transition table is the program of the parser. Because the transition table has a more complex structure, we need some more explanations before we discuss the details.*

For fully defining the transition table, we need two further sets which are given by the above parts of the automaton:

First of all,

$$L = \Sigma^{<k} \cdot (\Sigma \cup \{\$\})$$

is the *set of possible lookahead strings* (which does not include the empty word) and

$$A = \{\text{shift}(q), \text{goto}(q) | q \in \Gamma\} \cup \{\text{reduce}(r) | r \in P\} \cup \{\text{accept}, \text{error}\}$$

is the *set of actions.*

The transition table stores the information which allows the parser to choose the correct next step in a computation. It is built off two parts:

1. The shift/reduce-part uses the top of the stack and the lookahead from the position of the input head to determine whether a shift- or a reduce-operation is necessary. Whenever a shift is needed that means, some more information is put onto the stack. On the other hand, when a reduce-operation is necessary, enough information has been collected on the stack in order to identify a rule of the grammar which was applied. This results in removing the part of the stack, which corresponds to the left-hand side of a rule.
2. After each reduce-operation, a goto-operation checks the top of the stack and pushes another symbol onto it which corresponds to the information still in the stack and to the left-hand side of the respective rule.

After these short introduction, we now give the full definition of the transition table: T is a relation

$$T \subseteq (L \cup N) \times \Gamma \times A.$$

Those elements of T that have a word from L in their first component belong to the shift/reduce-part, whereas those with an element of N belong to the goto-part.

On T, the following restrictions are imposed:

(i) $(l, s, a) \in T$ and $l \in L$
$\Longrightarrow \nexists s' \in \Gamma : a = \text{goto}(s')$.

(ii) $(x, s, a) \in T$ and $x \in N \cup \{\Box\}$
$\Longrightarrow a = \text{error} \vee \exists s' \in \Gamma : a = \text{goto}(s')$.
(iii) $(l_1, s, a_1), (l_2, s, a_2) \in T; l_1, l_2 \in L; l_1 \in \text{Pref}(l_2)$
$\Longrightarrow l_1 = l_2 \wedge a_1 = a_2$.
(iv) $(x, s, a_1), (x, s, a_2) \in T; x \in N$
$\Longrightarrow a_1 = a_2$.
(v) $(\$, s, a)$
$\Longrightarrow \nexists s' \in \Gamma : a = \text{shift}(s')$
(vi) $\forall l \in L, s \in \Gamma : \exists l' \in \text{Pref}(l), a \in A : (l', s, a) \in T$.
(vii) $\forall x \in N, s \in \Gamma : \exists a \in A : (x, s, a) \in T$.

By the first two restrictions, the table is split into two parts. One part is for lookaheads, it may contain shift, reduce, accept, and error actions. The other part is for nonterminals of the grammar, it may contain goto and error actions.

The third and fourth restriction ensure that the automaton is deterministic.

The fifth restriction prevents the end marker from being shifted onto the stack, as given in the following definition of the computations possible with M_G.

The last two restrictions ensure that the behavior of the automaton is defined in any situation. In a certain sense they make the transition table complete. Usually this is obtained by filling up the table with error actions.

Definition 5.2. *The set C of* configurations of M_G *is defined as follows.*

$$C = K^+ \Sigma^* \$ \cup \{\mathit{accept}, \mathit{error}\}.$$

The K^+ part is the stack *of the configurations, the Σ^* part is the input not yet consumed. The top of the stack is also the state of the automaton.*

The initial configuration *for an input word $w \in \Sigma^*$ is $(s_0, \Box) \cdot w\$$. Except for halting configurations, where the state always is accept or error these two states will not be part of the configuration.*

Definition 5.3. *Single computation steps of the automaton are denoted with the relation $\underset{M_G}{\vdash}$. Let $c = uqv$ be a configuration with $u \in K^*, q \in K$,*

$v \in \Sigma^*\$$. The next configuration c', $c \underset{M_G}{\vdash} c'$ is given by the following rules:

We may assume $q = (s, x)$ with $s \in \Gamma, x \in \Sigma \cup N \cup \Box$. Note that $x = \Box$ only will occur if $u = \Box$ and $q = (s_0, \Box)$, which only happens at the beginning of the computation.

By condition (vi) and (vii) there exists a prefix l of v such that $(l, s, a) \in T$. Then the action of M_G is defined by a.

(a) If $a = \text{shift}(s')$. Then the first letter of v is an element of Σ, that is, $v = bv'$ with $b \in \Sigma$ and suitable v'. This letter and s' are shifted onto the stack, that means

$$c' = u \cdot (s', b) \cdot v'.$$

(b) If $a = \text{reduce}(r)$. Then r is a rule from P. Let n be the length of its right-hand side and $z \in N$ its left-hand side. In that case, by the construction of M_G from G (for example, see [Har78]) we know

$$c = u' \cdot (s', y) \cdot u''v, (s', y) \in K, |u''| = n.$$

Therefore, there is an $a' \in A$ with $(y, s', a') \in T$. If $a' = \text{error}$, then $c' = \text{error}$, the input is rejected. Otherwise, $a' = \text{goto}(s'')$. Then the next configuration is:

$$c' = u' \cdot (s', y) \cdot (s'', z) \cdot v.$$

That means, a reduction by rule r takes place. For further use, we refer to this as *reduce/goto procedure.*

(c) If $a = \text{accept}$, the input is accepted, $c' = \text{accept}$, and the automaton stops computation after that (no next configuration is defined).

(d) If $a = \text{error}$, the input is rejected, $c' = \text{error}$, and the automaton stops computation after that.

The reflexive, transitive closure of $\underset{M_G}{\vdash}$ is $\vdash^*{}_{M_G}$, and $L(M_G)$, the language accepted by it is defined as:

$$L(M_G) := \{w \in \Sigma^* | s_0 w\$ \vdash^*{}_{M_G} \text{accept}\}.$$

Because M_G is a parser especially constructed for G it is furthermore given that $L_G = L(M_G)$ and that it always either halts in the accept or the error configuration [Har78].

The explicit use of an end marker is a technical detail which may be confusing, since for $k > 0$ the deterministic context-free languages need not be prefix-free. Actually, the class of \$LR(k)-grammars are hidden behind that. For details we refer the reader especially to [GH77].

Remark 5.1. In the literature, the elements of Γ are called *states* very often. This is done because they are computed as states of a deterministic finite automaton. But in the usual understanding of automata theory, M_G has three states, that are a computation state, the accept state, and the error state. Therefore, we follow [Har78] and call the elements of Γ *tables*, although this might be confused with the *transition* table.

The connection to CRL

The next two theorems link this definition to automata that are used in the context of our investigation:

Theorem 5.1. *For every DCFL L there exists an sDTPDA M such that $L = L(M)$.*

Proof. There exist an $LR(k)$-grammar $G = (\Sigma, N, S, P)$ with $k \geq 1$ and $L_G = L$ and a shift/reduce-parser M_G for G with $L_{M_G} = L$. We may assume that S does not appear on the right-hand side of a rule in P and that the empty word does not appear on the right-hand side of a rule [GH77].[1] Furthermore, we may assume that P contains no cycles of chain rules.[2] Then G is also an acyclic context-sensitive grammar and therefore is weight-increasing, too [Bun96]. This will be used to obtain a shrinking DTPDA.[3]

[1] Except for a single rule $S \to \square$ if the empty word is in L.

[2] G would be ambiguous, otherwise, and therefore not an LR(k)-grammar [GH77].

[3] If $\square \in L$ this needs minor adjustments which we do not discuss because they are obvious.

For the sake of simplicity we allow the sDTPDA M to have a program δ which maps a finite $D \subset (Q \times \Gamma^+ \times \Gamma^+)$ into $(Q \times \Gamma^* \times \Gamma^*)$. Since sDTPDAs characterize CRL this is not too powerful a generalization.

The rest is only a matter of comparing definitions. As can be seen right-away, the main differences are the notation of the program and the way of accepting words.

Some efforts are necessary to simulate the behavior of reduce/goto actions. Basically, there are two possibilities: When M simulates a reduce action for a rule r with n being the size of its left-hand side, it may use a set of states to delete $n-1$ symbols from the stack one after the other (regardless what these symbols are), and then replaces the last symbol by the correct table/nonterminal pair. On the other hand, one can simulate the complete reduce/goto procedure in a single step. For that purpose it is not necessary to consider all possible strings of length n: There is only a (usually rather small) subset of those strings that can appear on top of the stack when the reduce/goto procedure has to be applied.

The question of weight reduction is quite easy. Mainly, one has to assure that all stack symbols have the same weight as the nonterminals which they contain, that all terminals have the same weight, and that the former is smaller than all weights of the latter, multiplied with a constant $c > 0$ if necessary. The weight of the states of M then can be easily found, because most of the time M will be in its initial state, and only when accepting or rejecting the state is changed. □

Remark 5.2. Shift/reduce-parsers only accept when the stack contains the initial (and bottom) stack symbol $(s_0, \Box)$ plus a symbol signifying the start symbol S and when the input is fully read. So it is possible to guarantee that the constructed sDTPDA accepts with final state and empty stacks.

The next corollary restates the inclusion part of theorem 2.18:

Corollary 5.1. *Each deterministic context-free language is a CRL.*

Here we see that in contrast to [MNO88] the proof of inclusion does not need the existence of LR(k)-grammars in Greibach normal form.

This is a direct consequence of the fact that length reduction and weight reduction are equally powerful for CRL.

Theorem 5.2. *Let $G = (\Gamma, N, S, P)$ be an LR(k)-grammar, $k \geq 1$, without cycles of chain rules, and with neither S nor $\Box$ appearing on the right-hand side of a rule in P. Then there is a csCRLS C with $L_C = L_G$. Furthermore, it is possible to construct C without the normal form construction of theorem 3.1.*

Proof. The program of the sDTPDA as defined above directly leads to the rewriting rules of C. It is clear that shift rules can be simulated by context-splittable rules. We may assume that reduce/goto-procedures are done with one single computation step, thus implying that the corresponding rules are context-splittable. Because of remark 5.2, we know that s_0 and \$ are removed in the last step, and that then the automaton is in a special accept state. This delivers the rest of the conditions for context-splittability.

Confluence is guaranteed by the determinism of the shift/reduce-parser and by the structure of the configurations. $\Box$

Remark 5.3. We wish to point out that—because we do not use theorem 3.1—the structure of the underlying shift/reduce-parser and the respective LR(k)-grammar are strongly linked to the constructed csCRLS.

Theorem 5.3. *Let $G = (\Sigma, N, S, P)$ be an LR(k)-grammar without cycles of chain rules, and with neither S nor $\Box$ appearing on the right-hand side of a rule in P. Assume C is a csCRLS C with $L_C = L_G$, constructed as in the last theorem. Assume there is a weight function φ such that*

$$\forall A \in N, x \in (N \cup \Sigma), w \in (N \cup \Sigma)^*, r = (A, xw) \in P : \varphi(x) > \varphi(A).$$

Then the rules given by the prefix system construction applied on C are all weight reducing.

Proof. One can give the terminals of C an arbitrarily high weight, therefore the shift rules and their prefix rules do not cause conflicts. The only possible conflicts with weight reduction would come from

the reduce/goto rules of the automaton. From the construction of the shift/reduce-parsers one knows that every reduce computation is closely linked to rules in P: There is a projection h from K to N which can be extended to a morphism from K^* to N^* such that for all $xul(x \in K, u \in K^+, l \in L)$ which are replaced by $xzl(z \in K)$ with a rule simulating a reduce/goto procedure the following holds:

$$(h(z), h(u)) \in P$$

and applying h to all reduce steps in a configuration in that manner gives a rightmost parse when read backwards. The projection simply is defined by:

$$h((q, x)) := x \text{ for } (q, x) \in K.$$

In consequence, the condition on the weights of the first symbols of each right-hand side of a rule in comparison to the left-hand side can be used. We define the weight function for pushdown symbols of the sDTPDA:

$$\psi(K) = c \cdot \varphi(h(K)),$$

where $c > 0$ is a constant to adjust weights if necessary. The weight for the terminals can be chosen to be higher than the maximum of these weights. □

Remark 5.4. The conditions of the theorem imply that there is no left recursion in the grammar. Again, this reminds of the problem of Greibach normal form for LR(k)-grammars (see also [GHH76]).

The next question to be considered is that of confluence of prefix systems. There are three possible types of critical pairs:

1. From two shift rules,
2. from two reduce rules, and
3. from one shift and one reduce rule.

We only consider the first case. Assume $u \in \Gamma^+, u', u'' \in \text{Suff}(u)$, $v \in \Sigma^+, v', v'', v''' \in \text{Pref}(v), x \in \Sigma, Q', Q'' \in \Gamma$.

Now there can be shift rules in the parser which can be applied to a full input. The corresponding rewriting rules have to look like the following:

$$(u'xv', u'Q'v') \text{ and } (u''xv'', u''Q''v'').$$

We look at possible overlapping prefix rules:

$$(u'xv''', u'Q'v''') \text{ and } (u''xv''', u''Q''v'''),$$

where $v' = v'''c'$ and $v'' = v'''c''$. Since we did not make any further assumptions about the shift/reduce-parser, both $u'Q'v'''$ and $u''Q''v'''$ can be irreducible. Then they are a critical pair which can not be resolved.

Similar conflicts can happen in the other two cases.

Conclusion

In this section, we have seen, that it is possible to construct a context-splittable CRLS directly from the automaton accepting a deterministic context-free language. As a consequence it is not necessary to use the construction of chapter 3. On the other hand, CRL and DCFL do not have so much in common that it can be guaranteed that the prefix systems constructed directly are correct. Although DCFL is closed against the prefix language (which makes it possible to find a CRLS for the prefix language), the prefix construction can lead to problems. We concentrated on the problem of confluence, because this is the more complicated necessary property one needs for correct prefix systems.

We conjecture that, if the prefix system is confluent, then it also is correct. An argument for this conjecture is that on the one hand DCFL is closed against building the prefix language and on the other hand shift/reduce-parsers find errors at the earliest possible moment during the parsing process. This should transfer to the prefix construction, therefore enhancing the possibility to handle completions correctly. Furthermore, computing a terminal completion for a correct nonterminal completion is possible with the help of the original $LR(k)$-grammar, if it is assumed that all nonterminals of the grammar can be derived to a terminal word (which of course is no unnatural restriction).

Finally, we believe that the connection between CRL and DCFL and their respective prefix languages can be exploited in order to gain further insight into DCFL.

5.2 A Development Environment

Understanding the practical properties of a language class like the Church-Rosser languages needs experiments. So, while the main focus of this dissertation are theoretical results, an integrated development environment which can be used for such experiments is described in this section.[4]

All results about prefix languages of Church-Rosser languages described in the last chapter have their roots in a prototype system written in LISP. This system delivered the means to describe CRLSs in a concise way and implemented the most basic algorithms for rewriting: the REDUCE-algorithm, testing for weight reduction, and testing for (local) confluence. Furthermore, in this prototype the first experimental algorithms for Church-Rosser prefix language systems appeared. They were the basis of all theoretical results concerning prefix systems discussed in the earlier chapters.

Soon the need for a more user friendly environment arose. The core specification was obvious: Implementation of all algorithms that were already present in the LISP-prototype, enriched by a means to load, edit, and save descriptions of Church-Rosser languages. Based on this specification, Rottschäfer implemented the first version of the Church-Rosser development environment which is described in his diploma thesis [Rot00]. One of the main (and most time consuming) goals of his work was to use this environment for giving a description of JAVA as Church-Rosser language and to test whether the prefix system computed for this is usable (from a more practical point of view) or not.

[4] J.-M. Kuhnigk developped a system for general GCSLs and TPDAs, called TPDAsim. Since the scope of this is somewhat different, we only give a reference here: http://www.tcs.mu-luebeck.de/pages/students/kuhnigk/Studienarbeit.html

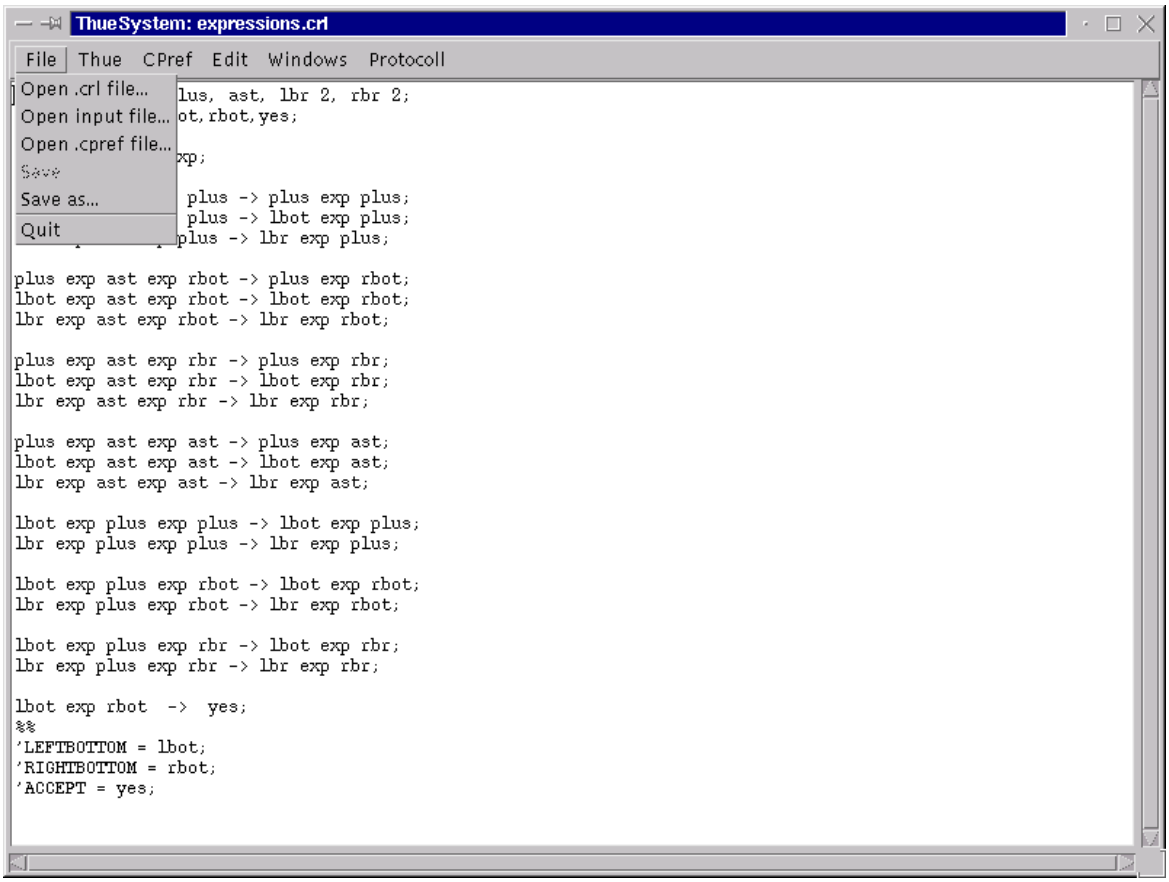

Fig. 5.1: The main window of the environment

As it is typical for bigger software projects, this environment is the result of a joint effort. The main responsibilities of the author of this dissertation were the specification of the system and supervising the correctness and usability of it. On the other hand, the coding (using JAVA) was performed by Rottschäfer. Therefore, we will not discuss implementation details too deeply here, the reader is referred to [Rot00].

Because of the experimental—and prototypical—character of the program, its development will be continued. In [Rot00], its first development phase is described, but in the meantime further work has been done. Except for the changes of the "Look and Feel", we will point out these changes at the appropriate places.

Editing features

The graphical user interface supplies the usual options for editing, saving and reloading files. This includes Church-Rosser language definitions (`.crl`-files) and input for the Church-Rosser parser (`.inp`-files). Furthermore, prefix systems computed by the program (`.cpref`-files) can be loaded. Figure 5.1 shows a typical view of the main window.

Defining Church-Rosser languages

An example for the definition of a Church-Rosser language is given in figure 5.2. These descriptions (saved in `.crl`-files) contain three parts:

```
% token section
'terminal a 2, b, c, d 2, e 2;
'non terminal lbot, rbot, y;
%%
% rule section
a b c -> b ;
b b rbot -> b rbot ;
d b -> b ;
e c -> b c ;
a b b b c -> a b c ;
lbot b rbot -> y ;
%%
% marker section
'LEFTBOTTOM = lbot ;
'RIGHTBOTTOM = rbot ;
'ACCEPT = y ;
```

Fig. 5.2: An example for a Church-Rosser language definition

1. The *token*[5] *section*, which defines the terminal and nonterminal alphabets of a CRLS,

[5] We use the terminology of scanning and parsing as used by standard tools like LEX and YACC.

2. the *rule section*, which contains the rewriting rules, and
3. the *marker section*, which defines the left-hand and right-hand bottom token (since this is no restriction of generality, the system allows only single tokens), and the accepting token.

The correspondence to the six parts of a CRLS, as given in definition 2.41 should be clear. A BNF-definition of these files is given in figure 5.3. The semantics of the `.crl`-files are as follows:

The token section

Since we want to use symbols that consist of more than one letter, there are two lexical analyzers available at the moment which were implemented using JLex: A scanner for Java tokens and a scanner for tokens consisting of letters of the alphabet {`a-z,A-Z,0-9,_`}. Figure 5.2 uses tokens of the latter.

All tokens used in the other two sections have to appear in the token section. This consists of two parts. First, all terminals have to be defined. This definition consists of a "`'terminal`" and a comma separated list of tokens, closed by a semicolon. For better readability it is also possible to use more than one of those lists. Furthermore, to each token a weight (a natural number) can be attached. If no weight is given, a default of 1 is assumed. The definitions of nonterminals following after that are similar, they start with "`'non terminal`". The example also shows comments, they begin with a `%` and end at the end of a line.

Remark 5.5. The distinction between terminals and non terminals is only a means of "commenting" the description, it is neither checked in the other two sections, nor used by the Church-Rosser parser, except for testing terminal correctness of prefix systems.

The rule section

The most simple way of defining a rule is just to specify the left-hand and right-hand side of it as a lists of tokens, divided by a "`->`". Additionally it is possible to quantify sets of rules, using variables. This is not shown in figure 5.2, so we give another example:

Example 5.1. (Quantifying rules)

```
$left exp $op exp $right -> $left exp $right|
        $left  = [ name assign, lbr, plus ]
        $op    = [ plus ]
        $right = [ semi, rbr, plus ];
```

It is possible to use *variables* which are substituted by (lists of) tokens. Using the generic scanner, variables start with a "$", with the JAVA-scanner this special character is not needed at the beginning.

If after the right-hand side of the rule there is a "|", then a list of variable substitutions follows. Each substitutions assign a comma separated list of token lists to a variable.

All combinations of possible substitutions are generated. In the example that would lead to nine rules, some of these are:

```
name assign exp plus exp semi -> name assign exp semi
name assign exp plus exp plus -> name assign exp plus
lbr exp plus exp rbr -> lbr exp rbr
plus exp plus exp plus -> plus exp plus
```

Remark 5.6. Although it is not explicitly forbidden to substitute variables by others variables it is not advisable to do so, because the behavior is not specified in that case.

There is another difference between the generic and the JAVA-scanner: The former uses square brackets, the latter curly brackets for enclosing the substitution list.

A special feature which can only be used with the generic scanner is derived from the ambiquity of context-splittings: Sometimes it is useful to give *explicit split points* for rules of a csCRLS instead of relying on the system to identify them. These split points are marked by a ":", like in the following example:

```
assign : exp plus exp : semi -> assign : exp : semi ;
```

Explicit split points are a new feature which was not present in the original environment.

```
CRLS            ::= TOKEN_SECT "%%" RULE_SECT "%%" MARKER_SECT

TOKEN_SECT      ::= TERMS NON_TERMS
TERMS           ::= TERM_LIST | TERM_LIST TERMS
NON_TERMS       ::= NON_TERM_LIST | NON_TERM_LIST TERMS
TERM_LIST       ::= " 'terminal" TOKEN_LIST
NON_TERM_LIST   ::= " 'non terminal" TOKEN_LIST
TOKEN_LIST      ::= TOKEN_DEF ";" | TOKEN_DEF "," TOKEN_LIST
TOKEN_DEF       ::= TOKEN | TOKEN WEIGHT

RULE_SECT       ::= RULE | RULE RULE_SECT
RULE            ::= RULE_SIDE "->" RULE_SIDE ";" |
                    RULE_SIDE "->" RULE_SIDE "|" SUBST_LIST ";"
RULE_SIDE       ::= RULE_TOKENS |
                    (with generic scanner also:)
                    RULE_TOKENS ":"RULE_TOKENS |
                    RULE_TOKENS ":"RULE_TOKENS ":"RULE_TOKENS
RULE_TOKENS     ::= TOKEN_OR_VAR | TOKEN_OR_VAR RULE_TOKENS
TOKEN_OR_VAR    ::= TOKEN | VARIABLE
SUBST_LIST      ::= SUBST | SUBST SUBST_LIST
SUBST           ::= VARIABLE "=" L_BRACKET SUBST_WITH R_BRACKET
SUBST_WITH      ::= RULE_TOKENS | RULE_TOKENS "," SUBST_WITH

MARKER_SECT     ::= LBOTTOM RBOTTOM ACCEPT
LBOTTOM         ::= " 'LEFTBOTTOM" "=" TOKEN ";"
RBOTTOM         ::= " 'RIGHTBOTTOM" "=" TOKEN ";"
ACCEPT          ::= " 'ACCEPT" "=" TOKEN ";"

TOKEN           ::= ...(as defined by scanner)
WEIGHT          ::= ...(natural number, defined by scanner)
VARIABLE        ::= ...(as defined by scanner)
L_BRACKET       ::= ...("{" or "[", as defined by scanner)
R_BRACKET       ::= ...("}" or "]", as defined by scanner)
```

Fig. 5.3: The BNF for Church-Rosser language definitions

Basic operations

When the program reads a `.crl`-file, or when the user has changed it and tells the program to do so, from the definition in the file a rewriting system is generated as an internal representation.

Using the internal representation, the following basic operations are possible with the rewriting system:

Testing for weight reduction: Based on the weight given in the `.crl`-file the program can check if all rules of a rewriting system are weight reducing.

Reducing an (input) word: With the program an input file (`.inp`-file) can be loaded and edited. This input may consist of the tokens defined in the token section of the `.crl`-file. The same scanner that is used for reading the `.crl`-file is used for the `.inp`-file.

This input can be reduced by the program with the standard REDUCE-algorithm[6]. The implementation uses a hash to find the left-hand rule sides which can be applied. That same algorithm can be used for internal representations of words, e.g. for testing local confluence:

Testing for local confluence: Rewriting systems can be tested for local confluence with the CONFTEST-algorithm (definition 2.33). First, a check for weight reduction is performed, since otherwise termination of the REDUCE-algorithm could not be guaranteed. Figure 5.4 shows the output of such a confluence test.

Generating and testing prefix systems

After a Church-Rosser language system has been generated (and successfully checked for weight reduction and confluence), the following further operations can be performed:

[6] The original code, as written by Rottschäfer, has been slightly improved by the author of this dissertation. Most of these changes concerned the logical structure of the code, in order to make it both more readable and closer to the theoretical basis of the algorithm.

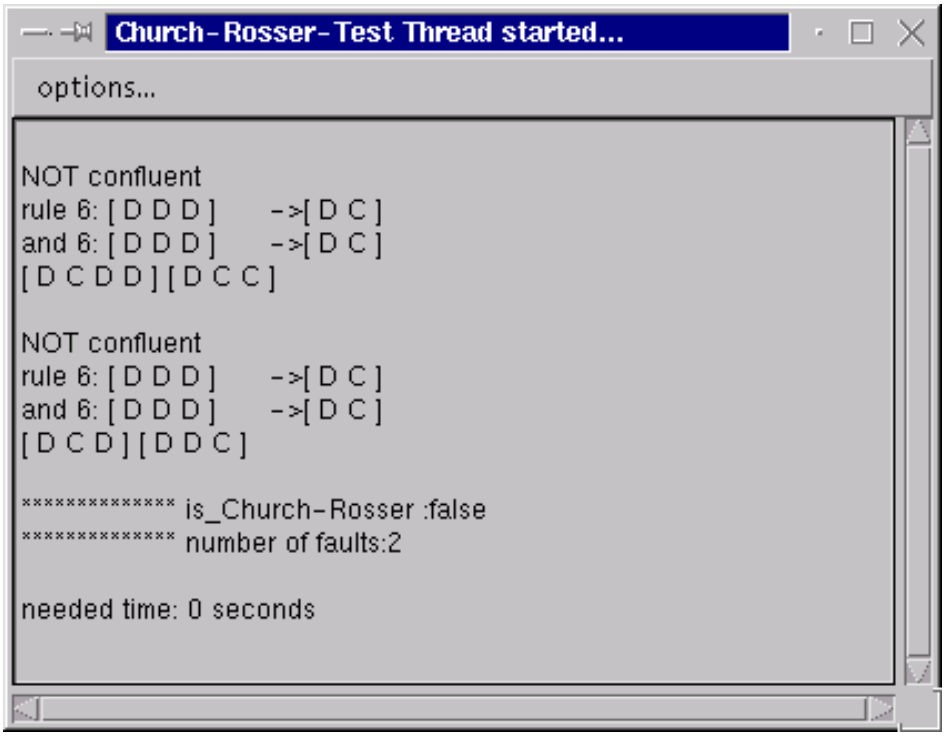

Fig. 5.4: A confluence test (rejected)

Generate a prefix system: A (completion) prefix system, as described in the previous chapter, is generated. The rewriting system replaces the original rewriting system. Furthermore, an internal representation of the completion prefix system is generated, which can also be displayed.

Generate a suffix system: The symmetrical procedure is used to build a rewriting system for the suffix language. In this case, no new completion system is generated.

Generate an infix system: First, an internal suffix system is built, and from that a prefix system and a completion prefix system is computed. The completions of the latter can be displayed.

The last two options are only implemented for use with the reduce procedure, the following procedures do not make sense for them within the environment:

Compute valid set: The program computes the candidate set, as given in definition 4.11. Since this procedure only terminates if this

candidate set is finite, it is possible to abort this computation. Furthermore, if the computation lasts very long, from time to time a dialog appears which points out that the computation possibly should be aborted, lest the user thinks the program has "'hung up"'.
After computation of the candidate set, it is checked if the completion prefix system is correct in the sense of theorem 4.6.
The implementation of this feature was modified by the author of this dissertation.

Extended valid sets: This is an experimental feature which is similar to the ideas of theorem 4.11, in fact it inspired the theorem, but is less powerful.[7]
This is a new feature which was not in the scope of [Rot00].

Testing terminal correctness: According to theorem 4.13 terminal correctness of the completion prefix system is checked. If there are only finitely many nonterminal completions for basic valid reductions, these are all checked whether they can be reached by a reduction from a terminal string. Since it is not known how many rewritings steps (the k in the theorem) are necessary, the system starts with one step and increases the number until either correctness can be shown or the user aborts the computation.
This is a new feature.

Using prefix systems

Prefix systems can be used on an input (.inp-file) in two ways:

Reducing the input: In order to determine whether an input is a valid prefix of the original language, a usual reduction with the prefix system can be executed.

Longest prefix completion: The program searches the longest correct prefix of an input with the prefix system. After doing that, it suggests possible completions to a correct word. The user can choose a completion in order to compute the reduction result that is achieved by first adding that completion. In figure 5.5 an example for this is given.

[7] For obvious reasons: testing correctness as in that theorem has far too big a time complexity.

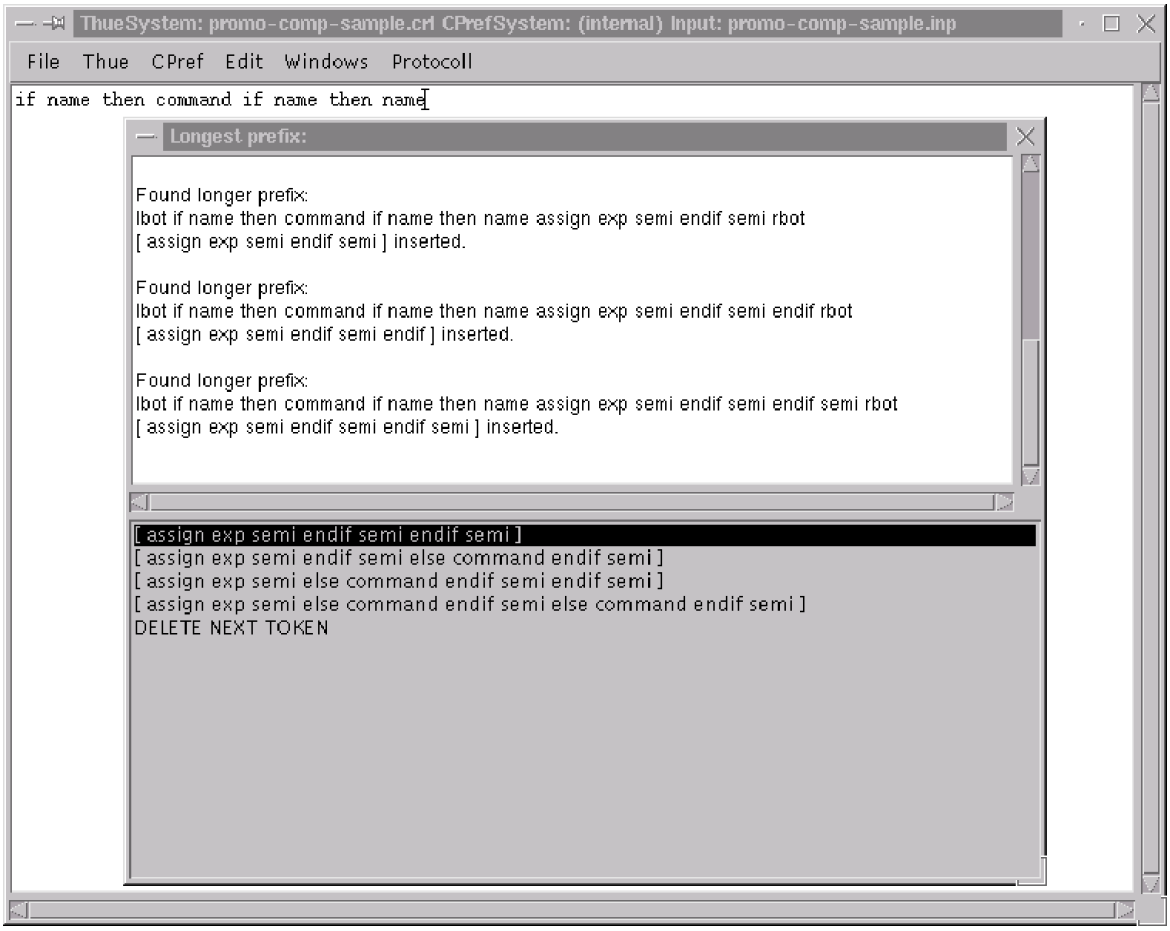

Fig. 5.5: Suggested completions for a prefix of a simple language

A note on the Java-CRLS

Rottschäfer showed in his diploma thesis that the important syntactic features of JAVA can be parsed with CRLSs whose prefix system is correct. An interesting result of his work is that the suffix language also can be accepted by suffix CRLSs, but in that case one has to use a different CRLS to define JAVA, because otherwise the suffix construction leads to a system not confluent. This seems to reflect the fact that DCFL is not closed against building the mirror language. On the other hand, it shows the potential of CRL.

Chapter 6

Conclusion

In the article in which McNaughton, Narendran, and Otto define the Curch-Rosser languages, they state the opinion:

> We think that CR systems will provide an important systematic approach to string computation, although we do not think they will improve on the current techniques for writing compilers.
> [MNO88]

Considering the results established since 1988, especially in connection to growing context-sensitive languages, the first part of their prediction surely has become true. Some of the techniques developed in the context of Church-Rosser languages, like that of Niemann and Otto with which they proof that length reduction and weight reduction are equally powerful, are at least interesting from the theoretical point of view. We are confident that both the method of weight-spreading and the prefix construction described in the preceding chapters are equally interesting.

From the more practical point of view, we believe that Church-Rosser languages are more interesting than they appear at first when taking a closer look. We consider it as promising that our prefix construction does not only provide a system for the prefix language, but also makes it possible to generate completions to correct words in a very natural way. In connection with shift/reduce-parsers, maybe in a hybrid system combining both worlds, this might allow more comfortable error correction methods.

Appendix

List of Figures

Bibliography

[App98] A. W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[AU72] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling. Volume I: Parsing.* Prentice-Hall, 1972.

[AU73] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling. Volume II: Compiling.* Prentice-Hall, 1973.

[BF87] M. G. Burke and G. A. Fisher. A practical method for LR and LL syntactic error diagnosis and recovery. *ACM Transactions on Programming Languages and Systems*, 9(2):164–167, 1987.

[BHNO00a] M. Beaudry, M. Holzer, G. Niemann, and F. Otto. McNaughton languages. In R. Freund, editor, *Theorietag 2000 mit Workshop New Computing Paradigms: Molecular Computing and Quantum Computing*, pages 159–165. Technische Universität Wien, 2000. ISBN 3-85028-325-9.

[BHNO00b] M. Beaudry, M. Holzer, G. Niemann, and F. Otto. McNaughton languages. Technical Report Mathematische Schriften Kassel, No. 26/2000, Universität Kassel, November 2000. Full version of [BHNO00a].

[BL92] G. Buntrock and K. Loryś. On growing context-sensitive languages. In W. Kuich, editor, *Proc. of ICALP 1992 (International Conference on Automata, Languages, and Programmin)*, volume 623 of *LNCS*, pages 77–88, Berlin, 1992. Springer-Verlag.

[BL94] G. Buntrock and K. Loryś. The variable membership problem: Succinctness versus complexity. In P. Enjalbert, E.W. Mayr, and K.W. Wagner, editors, *Proc. of STACS 1994 (Symposium on Theoretical Aspects of Computer Science)*, volume 775 of *LNCS*, pages 595–606, Berlin, 1994. Springer-Verlag.

[BO81] R. V. Book and C. O'Dunlaing. Testing for the Church-Rosser property. *Theoretical Computer Science*, 16:223–229, 1981.

[BO84] G. Bauer and F. Otto. Finite complete rewriting systems and the complexity of the word problem. *Acta Informatica*, 21:521–540, 1984.

[BO93] R. V. Book and F. Otto. *String-Rewriting Systems.* Springer-Verlag, New York, 1993.

[BO95] G. Buntrock and F. Otto. Growing context-sensitive languages and Church-Rosser languages. In E.W. Mayr and C. Puech, editors, *Proc. of STACS 1995 (Symposium on Theoretical Aspects of Computer Science)*, volume 900 of *LNCS*, pages 313–324, Berlin, 1995. Springer-Verlag. Extended abstract of [BO98].

[BO98] G. Buntrock and F. Otto. Growing context-sensitive languages and Church-Rosser languages. *Information and Computation*, 141:1–36, 1998.

[Boo69] R. V. Book. *Grammars with Time Functions.* PhD thesis, Harvard University, Cambridge, MA, February 1969.

[Boo73] R. V. Book. On the structure of context-sensitive grammars. *International Journal of Computer and Information Science*, 2:129–139, 1973.

[Boo78] R. V. Book. On the complexity of formal grammars. *Acta Informatica*, 9:171–181, 1978.

[Boo82] R. V. Book. Confluent and other types of Thue systems. *JACM*, 29:171–182, 1982.

[Bra74] F.-J. Brandenburg. Zur Verallgemeinerung von Grammatiken durch Kontext. Technical Report No. 73, Seminarberichte des Instituts für Theorie der Automaten und Schaltnetzwerke, Gesellschaft für Mathematik und Datenverarbeitung, Bonn, 1974.

[Bun96] G. Buntrock. *Wachsende kontext-sensitive Sprachem.* Habilitationsschrift, Fakultät für Mathematik und Informatik, Universität Würzburg, Juli 1996.

[Cho59] N. Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137–167, June 1959.

[DW86] E. Dahlhaus and M.K. Warmuth. Membership for growing context-sensitive grammars is polynomial. *Journal of Computer and System Sciences*, 33:456–472, 1986.

[GG65] S. Ginsburg and S. Greibach. Deterministic context-free languages (preliminary report). *Am. Math. Soc. Not.*, 2:246,367, 1965.

[GH77] M. M. Geller and M. A. Harrison. On LR(k) grammars and languages. *Theoretical Computer Science*, 4:245–276, 1977.

[GHH76] M. M. Geller, M. A. Harrison, and I. Havel. Normal forms of deterministic grammars. *Discrete Mathematics*, 16:313–322, 1976.

[Gla64] A. V. Gladkij. On the complexity of derivations in context-sensitive grammars. *Algebra i Logika, Seminar*, 3(5–6):29–44, 1964. In Russian.

[Har78] M. A. Harrison. *Introduction to Formal Language Theory.* Addison-Wesley, Reading, MA, 1978.

[Hot66] G. Hotz. Eindeutigkeit und Mehrdeutigkeit formaler Sprachen. *Elektronische Informationsverarbeitung und Kybernetik (Journal of Information Processing and Cybernetics*, 2(4):235–246, 1966.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, Reading, MA, 1979.

[Jan88] M. Jantzen. *Confluent String Rewriting.* Springer-Verlag, 1988.

[Joh75] S. C. Johnson. Yacc – yet another compiler compiler. Technical Report CSTR-32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[KKMN85] D. Kapur, M. Krishnamoorthy, R. McNaughton, and P. Narendran. An $O(|T|^3)$ algorithm for testing the Church-Rosser property of Thue systems. *Theoretical Computer Science*, 35:109–114, 1985.

[Knu65] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8:607–639, 1965.

[McN99] R. McNaughton. An insertion into the Chomsky hierarchy? In J. Karhumäki, H. A. Maurer, G. Paŭn, and G. Rozenberg, editors, *Jewels are Forever, Contributions on Theoretical Computer Science in Honour of Arto Salomaa*, pages 204–212. Springer-Verlag, 1999.

[MNO88] R. McNaughton, P. Narendran, and F. Otto. Church-Rosser Thue systems and formal languages. *Journal Association Computing Machinery*, 35:324–344, 1988.

[Nie00] G. Niemann. Regular languages and Church-Rosser congruential languages. In R. Freund and A. Kelemenova, editors, *Proceedings of the International Workshop Grammar Systems 2000*, pages 359–370. Silesian University at Opava, Faculty of Philosophy and Science, Institute of Computer Science, 2000.

[Nij80] A. Nijholt. *Context-Free Grammars: Covers, Normal Forms, and Parsing.* Springer-Verlag, 1980.

[NO97] G. Niemann and F. Otto. The Church-Rosser languages are the deterministic variants of the growing context-sensitive languages. Technical Report GhK-FB 17: 8/97, Universität Kassel, 1997.

[NO98] G. Niemann and F. Otto. The Church-Rosser languages are the deterministic variants of the growing context-sensitive languages. In M. Nivat, editor, *Proc. of FoSSaCS 1998 (Foundations of Software Science and Computation Structures)*, volume 1378 of *LNCS*, pages 243–257, Berlin, 1998. Springer-Verlag.

[NO99] G. Niemann and F. Otto. Restarting automata, Church-Rosser languages, and confluent internal contextual languages. Technical Report Mathematische Schriften Kassel, No. 4/1999, Universität Kassel, 1999.

[NW01a] G. Niemann and J. R. Woinowski. The growing context-sensitive languages are the acyclic context-sensitive languages. In W. Kuich, editor, *Preproc. of DLT 2001 (Developments in Language Theory)*. Institut für Algebra und Mathematik, TU Wien, 2001.

[NW01b] G. Niemann and J. R. Woinowski. The growing context-sensitive languages are the acyclic context-sensitive languages. Technical Report GhK-FB 17: 01/2001, Universität Kassel, 2001.

[OKK97] F. Otto, M. Katsura, and Y. Kobayashi. Cross-sections for finitely presented monoids with decidable word problems. In H. Comon, editor, *Rewriting Techniques and Applications*, volume 1232 of *LNCS*, pages 53–67, Berlin, 1997. Springer-Verlag.

[Ott84] F. Otto. Some undecidability results for non-monadic Church-Rosser Thue systems. *Theoretical Computer Science*, 33:261–278, 1984.

[Par66] R. J. Parikh. On context-free languages. *Journal of the Association of Computing Machinery*, 13:570–581, 1966.

[Rot00] T. Rottschäfer. Eine Entwicklungsumgebung für Church-Rosser Präfixparser. Diplomarbeit, TU-Darmstadt, February 2000.

[Sal73] A. Salomaa. *Formal Languages.* Academic Press, New York, 1973.

[SSS90] S. Sippu and E. Soisalon-Soininen. *Parsing Theory. Volume II: LR(k) and LL(k) Parsing.* Springer-Verlag, Berlin, 1990.

[Woi00a] J. R. Woinowski. A normal form for Church-Rosser language systems. Technical Report TI-7/00, TU-Darmstadt, June 2000.

[Woi00b] J. R. Woinowski. A normal form for Church-Rosser language systems (abstract). In R. Freund, editor, *Theorietag 2000 mit Workshop New Computing Paradigms: Molecular Computing and Quatum Computing*, pages 223–227. Technische Universität Wien, 2000. (ISBN 3-85028-325-9).

[Woi00c] J. R. Woinowski. Prefix languages of Church-Rosser languages. In *Proc. FST-TCS 2000 (Foundations of Software Technology/Theoretical Computer Science)*, volume 1974 of *LNCS*, Berlin, 2000. Springer-Verlag.

[Woi00d] J. R. Woinowski. Prefixes of Church-Rosser languages. Technical Report TI-2/00, TU-Darmstadt, February 2000.

[Woi01] J. R. Woinowski. A normal form for Church-Rosser language systems. In Aart Middeldorp, editor, *Proc. of RTA 2001 (Rewriting Techniques and Applications)*, volume 2051 of *LNCS*, pages 322–327, Berlin, 2001. Springer-Verlag.

Abbreviations and Symbols

Index

This index contains references to the first appearance (except for the introduction chapter) or definition of the most important terms. For abbreviations and special symbols see the corresponding appendix on page 151.

Curriculum vitae

1969	born in Frankfurt am Main
1975-1980	Albert-Schweitzer-Grundschule, Hattersheim
1980-1986	Leibniz-Gymnasium, Frankfurt/Main
1986-1989	Friedrich-Dessauer-Gymnasium, Frankfurt/Main Allgemeine Hochschulreife/high-school graduate
1989-1990	Praunheimer Werkstätten Zivildienst/civil service
1990-1996	Technische Universität Darmstadt/ Darmstadt University of Technology Informatik-Studium/student of computer science Abschluss/graduated as: Diplom-Informatiker
1996-2001	Technische Universität Darmstadt/ Darmstadt University of Technology Fachgebiet Automatentheorie und Formale Sprachen/ Chair of "Automata Theory and Formal Languages" Wissenschaftlicher Mitarbeiter/assistant

Erklärung

Hiermit erkläre ich, die vorliegende Arbeit zur Erlangung des akademischen Grades Dr. rer.nat. mit dem Titel „Church-Rosser Languages and their Application to Parsing Problems“ selbständig und ausschließlich unter Verwendung der angegebenen Hilfsmittel erstellt zu haben. Ich habe bisher noch keine Promotionsversuche unternommen.

Darmstadt, 20. August 2001, Jens R. Woinowski