Reliable Confluence Analysis of Conditional Term Rewrite Systems

dissertation

by

Thomas Sternagel

submitted to the Faculty of Mathematics, Computer Science and Physics of the University of Innsbruck

in partial fulfillment of the requirements for the degree of Doctor of Philosophy

advisor: Univ.-Prof. Dr. Aart Middeldorp

Innsbruck, 4 August 2017

universität
innsbruck

dissertation

Reliable Confluence Analysis of Conditional Term Rewrite Systems

Thomas Sternagel (09916893)
thomas.sternagel@uibk.ac.at

4 August 2017

advisor: Univ.-Prof. Dr. Aart Middeldorp

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt durch meine eigenhändige Unterschrift, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Alle Stellen, die wörtlich oder inhaltlich den angegebenen Quellen entnommen wurden, sind als solche kenntlich gemacht.
Die vorliegende Arbeit wurde bisher in gleicher oder ähnlicher Form noch nicht als Magister-/Master-/Diplomarbeit/Dissertation eingereicht.

Datum	Unterschrift

Abstract

In a nutshell, this thesis lays the necessary groundwork to *automatically* and *reliably* check a certain property of computer programs written in a functional programming language. Because there are many different programming languages and we do not want to take their specific strategies and features into account we choose to work on an abstract *computation model* called *term rewriting*. Many results for basic term rewriting are already well-investigated and also formalized on the computer. Term rewriting is Turing-complete and hence capable of encoding any program, but plain term rewriting is not optimal in the area of functional programming because some naturally occurring constructs are somewhat cumbersome to represent within this formalism. For that reason we concentrate on a flavor of term rewriting that is better suited for our needs and is called *conditional term rewriting*. The property of conditional term rewriting we are most interested in is called *confluence* and it basically ensures that for a given input a program will always compute the same output even if it runs concurrently on distributed machines. This interesting property is *undecidable* in general, that means it is impossible to write an algorithm that given a program as input always outputs "yes" if the program is confluent and "no" if it is not. Sometimes we have to give up and admit that we do not know the answer. Within this thesis we first present the basics of conditional term rewriting and related topics and then investigate several known criteria that may be used to show confluence of a given program in the formalism of conditional term rewriting. Remember that our goal is to analyze programs automatically and reliably. Because of the former we also implemented all of the presented methods in an *automatic tool*, which can take a program as input and tries to decide if it is confluent or not. Concerning the reliability of our analysis we have to note that our automatic tool is also just a computer program, and a complex one for that matter, so it may very well contain bugs and maybe even give wrong answers. To tackle this problem we have *formalized* all of the above mentioned criteria in an interactive proof assistant, that is, we have scrutinized the methods and their proofs from the literature, filled in the gaps and ultimately provide such a level of detail, that the results are now computer-checkable by a relatively small and trusted program. Finally, with some additional effort, we are able to automatically generate another program from the computer-checkable *formalization*. This program is called a *certifier* and is able, as the name suggests, to ascertain if our first tool gave the correct answer by checking its output. To summarize, the thesis at hand contains a description of the underlying theory of our formalization concerning confluence of conditional term rewriting as well as a manual on how to use our tool in practice.

Acknowledgments

I thank my supervisor Aart Middeldorp, without whom this whole endeavor would not have been possible. I am especially indebted to my twin brother Christian Sternagel, who brought me into the fold in the first place. Without his patient guidance my formalization work would be rather short. Special thanks also goes to Harald Zankel for supervising my bachelor and master projects and always assuring me of doing a good job. Without his encouragement I may not even have entered the doctoral program. Furthermore, I want to thank the inmates of lab 3M03, especially Julian Nagele and Bertram Felgenhauer, for many profound discussions on the subject matter and badly needed coffee breaks. My family and close friends are very important to me and have certainly influenced my work for the better. I would like to thank them all, and especially my wife Kathrin, for their constant support. Thanks to Alicia, Cutter, and Harlon for providing much needed recreation and diversion. Finally, I want to thank the Austrian Science Fund for supporting my work through FWF international cooperation project I 963-N15 and FWF project P27502.

Contents

Chapter 1

Introduction

It is hard to imagine our modern world without laptops, smartphones, robots, spacecraft, computed tomography scanners, and so forth. Of course all of these complex systems rely on some kind of computers, which in turn are not very useful without software, that is programs that run on them, and provide actual services to their users.

Two properties that we would like to ensure for almost any program are:

1. to finish their computation within a finite amount of time.
2. to always produce the same output given the same input.

The former is called *termination* and, as we will see later on, is an important property to ensure correctness of a program. The latter, called *confluence*,[1] looks trivial on first sight but is of particular interest if a program runs in parallel or even on a distributed system, and hence the computing path can vary.

> The property that we investigate in this thesis is *confluence*.

Termination and confluence, like most interesting properties, are both instances of so called *undecidable decision problems*. That means, that it is impossible to construct a single algorithm that always leads to a correct yes-or-no answer. Still we can try to come up with procedures that can give answers for some interesting subset of programs.

Computers as well as the software that runs on them possibly contain errors, like for example *non-termination* and *non-confluence*. In the best case these errors never show up in practice, but more likely they will surface at some unfortunate point in time possibly causing substantial financial damage or even the loss of life. Even assuming that the computers that run the software are infallible, we still want to ensure the highest possible reliability of the software itself. However, for any non-trivial program testing can never exhaustively check all possible computation paths. To cite a famous computer scientist:

> *Program testing can be used to show the presence of bugs, but never to show their absence!*
>
> E. W. Dijkstra

Above all, because of the increasing parallelization of modern programs, testing is becoming more and more difficult. Hence testing alone is clearly not enough to check properties of safety-critical systems, instead we want to employ *formal verification*.

[1] Actually as stated this property is called *unique normal forms*, but it is implied by confluence.

Formal verification may be described in a few words as using methods from mathematics and logic, so called *formal methods*, to check some property of a given (software or hardware) system. In order to do that we first need some kind of formal description, not just in natural language but using logical formulas, of the system. This is usually called its *formal specification*.

Arguably the most interesting property of a system is its *correctness*, that is, the system is terminating and it does what it is supposed to do, according to its formal specification. Of course we can also employ formal verification to show other properties like confluence, for example. Leading hardware companies are already using formal verification since many years, unfortunately the software industry is still lagging behind. Probably because formal verification has a reputation to be too complicated and time consuming for most software projects. We believe that this is just a matter of tool support and our vision is to change that. To have any hope of widespread use, formal verification has to be usable by an average programmer, without doing a doctor's degree first. That most probably means that methods of formal verification have to be available in an *integrated development environment*, where the programmer, for example, just has to select some program code and can then automatically check properties like termination or confluence by the push of a button. That means, we need software tools that can check properties of programs automatically.

> The vision of this thesis is to provide *reliable* tools that *automatically* check properties of programs.

For the sake of argument assume we have a tool A that can check some property of a given program. Unfortunately we cannot trust tool A because ultimately it is just a computer program and may very well contain errors itself. So we need to prove it correct first. Since tool A is a complex piece of software its correctness proof could be difficult and what is more, whenever we extend or optimize tool A we have to redo this proof. To make matters worse, someone else might come up with a tool B that can check the same property as tool A but since it is written in another programming language and uses different algorithms, it needs its own correctness proof. Then again, we might be interested in checking some different property with a tool C, and again we need a new correctness proof for it, ...

This approach clearly does not scale too well, so we went for something different: We do not prove correctness of tools A to C at all, instead we develop a new tool, a so called *certifier*, that is able to rigorously assure correctness of a tool's output with respect to a given input. Now we only have to prove correctness of this certifier once and for all. Because doing a complex proof by hand is too error-prone we want to use a so called *proof assistant* to do the correctness proof for the certifier (see Figure 1.1). A proof assistant is a computer program that assists a human in establishing a formal and *computer-verified* proof. The human guides the proof construction by typing commands in some kind of *interactive proof editor*, while the assistant ensures that the proof is constructed only using logic inference rules from a small and trusted kernel. Trust in this part is founded on the fact that the kernel is small enough for several experts to have checked it and agreed that it is correct. Now, if the human succeeds in finishing

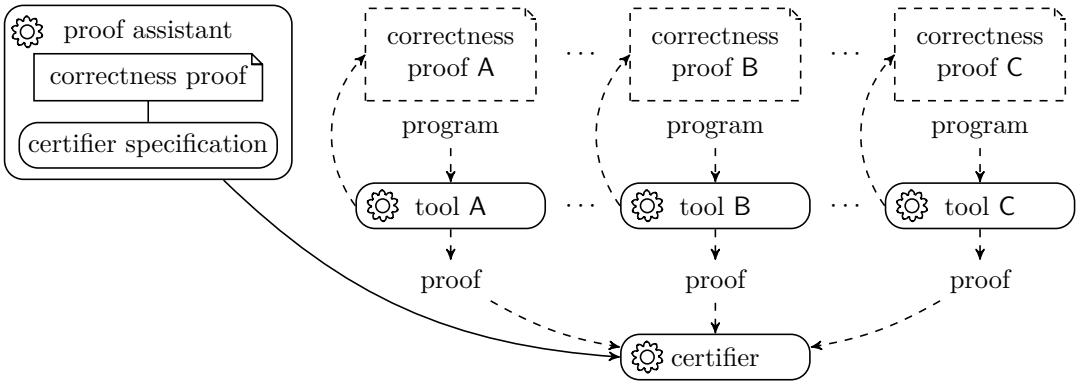


Figure 1.1: The certifier is proved correct in and then generated from a proof assistant.

the proof, it is not just formal, only using a small set of logic inference rules, but also *mechanized*, that is, a computer can check its correctness automatically (see Figure 1.2).

In summary, our approach to arrive at a reliable, automatic tool to check some property of a program involves:

1. coming up with and check the literature for algorithms and techniques to show a certain property and implementing them in a software tool,
2. taking the theorems about these techniques and formalizing them inside a proof assistant, thereby correcting possible errors, closing gabs, and providing technical details resulting in a library of formal definitions, theorems and proofs,
3. from this library automatically generating a certifier that is able to check if the algorithms and techniques to show the property have been applied correctly.

> Subsequently we will refer to these three steps as *automation*, *formalization*, and *certification*, respectively.

Now that we roughly know how to get an automatic tool to reliably check properties of some program, let's take a step back. How do the programs that we want to check, actually look like? We will first decide on a programming paradigm. Considering programming languages in general, there are basically two approaches to problem solving:

1. stating *how* to achieve something and as a result what should happen will happen.
2. focusing on *what* you would like to happen and letting the computer figure out how.

Programming languages of the former kind are called *imperative* and they typically feature things like while-loops, assignments, state-change, and side-effects. Prominent examples are Fortran, C, C++, and Java. In contrast, programming languages of the latter kind are called *functional-logic*. If used *purely* such languages do not have mutable state or side-effects and hence are well suited for parallel programming. Functional-logic languages have been considered to be only of academical interest in the past, but nowadays they are used for industrial and commercial applications. Well-known examples of functional-logic languages are Haskell, the Wolfram Language (underlying

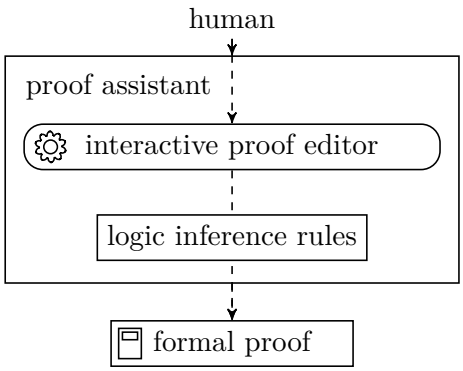


Figure 1.2: The process of formalizing a proof inside a proof assistant.

the computer algebra system Mathematica), Erlang (for example used in parts of WhatsApp), and Prolog.

For the rest of this thesis we will focus on *functional-logic programs*.

Having decided on the programming paradigm, there is still some ambiguity, in what we mean by "property of a program". Of course in the end we want to be certain that a program features a certain property *during execution*, but this depends on several factors besides the program itself. To really be able to formally verify a property of a program during execution, we would also need a verified compiler and assembler, as well as verified hardware and a verified operating system. Providing all of this goes beyond the scope of this thesis. As mentioned before, there are already hardware companies that use formal verification and more recently there are also verified compilers and operating systems emerging. So for the rest of this thesis we will in good conscience focus on verifying properties for functional-logic programs on the programming language level alone, and tacitly assume that everything below is done correctly and does not introduce new errors (see Figure 1.3).

Still, we do not want to take varying memory models and other specific features, like the evaluation strategy, of some concrete programming language into account, because this would be a nightmare to formalize. Instead we strive for a formal system that on the one hand is abstract enough to allow reasoning on a high level, but at the same time is also close enough to an actual programming language that we may express statements in a human-readable and succinct form. There are plenty of *models of computation* to choose from that fulfill the former requirement, but honestly it is not very convenient to work directly with Turing machines, lambda terms, while programs, register machines, or the like. Fortunately, the *Church-Turing thesis*, allows us to take any model of computation in order to abstract from a specific programming language. One model that is reasonably close to actual functional-logic programming is *term rewriting*.

Research in term rewriting is ongoing since over 30 years and nowadays there are already a number of tools in existence which allow us to conveniently check various properties of standard term rewrite systems. Also the formalization of standard term

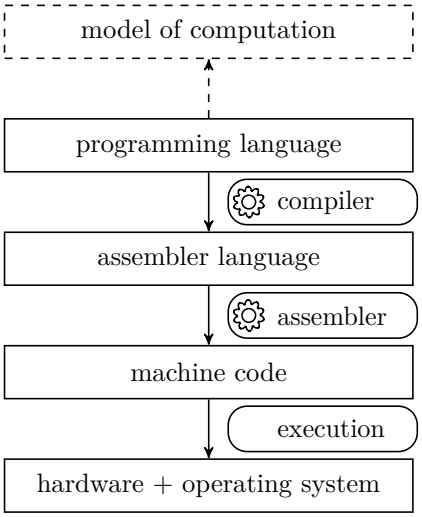


Figure 1.3: The usual software stack together with a model of computation.

rewriting is ongoing work since almost a decade, with many widely used results and so to not just rely on the trustworthiness and programming-prowess of the tool-authors, the above mentioned tools are progressively accompanied by certifiers.

Unfortunately, some typical features of functional-logic programming languages are cumbersome to represent in standard term rewriting. For that reason we will consider a flavor of term rewriting that has the same computing power while being more expressive.

The model of computation we use is called *conditional term rewriting*.

As an example, consider the Haskell program depicted in Figure 1.4(a). It consists of three equations that together define the minimum of a given list of natural numbers. In contrast, Figure 1.4(b) shows a straightforward translation of this Haskell program into a conditional term rewrite system. Here, instead of equations we have so called rewrite rules, consisting of a left-hand side and a right-hand side, separated by an arrow. The semantics is simple: whenever we encounter something similar to the left-hand side of some rule, we may replace it by the corresponding right-hand side. By doing this repeatedly we can compute the unique result of an input with respect to the conditional term rewrite system (if it is terminating and confluent, that is). Note that some of the rules are equipped with preconditions, these rules may only be applied if the conditions are fulfilled.

1.1 Contributions

To be slightly more concrete our contributions described in this thesis are the following:

Automation. We have implemented an automatic tool named ConCon that is able to check confluence (and some other properties) of programs represented as conditional

```
min (x:[])             = x
min (x:xs) | x < y     = x
           | otherwise = y
           where y = min xs
```

(a) Haskell.

$$\mathsf{min}(x : \mathsf{nil}) \to x$$
$$\mathsf{min}(x : xs) \to x \Leftarrow \mathsf{min}(xs) \approx y,\ x < y \approx \mathsf{true}$$
$$\mathsf{min}(x : xs) \to y \Leftarrow \mathsf{min}(xs) \approx y,\ x < y \approx \mathsf{false}$$

(b) Conditional term rewrite system.

Figure 1.4: Two versions of the same program.

term rewrite systems.

Formalization. We have used the proof assistant Isabelle/HOL to formalize the underlying theory and thereby extended the already existing formal library IsaFoR with several results on conditional term rewriting and related topics. Concretely these results are:

- a confluence method employing orthogonality,
- a critical pair criterion,
- tree automata techniques to help ignore certain cases of the confluence analysis,
- methods to find witnesses for non-confluence,
- methods to ignore or simplify rules of a conditional term rewrite system, and
- auxiliary definitions and lemmas to formalize all of the above.

Certification. Certification is really a joint effort by the tool that outputs a proof and the certifier that reads it. For that reason we extended both ConCon and IsaFoR (some more) in such a way, that the certifier CeTA,[2] that is automatically generated from IsaFoR, is now able to check all of the above mentioned techniques and at the same time ConCon can provide detailed enough output for all methods it implements in a format readable by CeTA.

Experiments. Our tool ConCon is taking part in the annual confluence competition – CoCo.[3] There automatic confluence tools test their mettle in several different categories. The problems that these tools try to solve come from the confluence problems database – Cops[4] (version 542 at the time of writing). Beyond the results we get from this competition we provide extensive experiments, comparing the different methods of ConCon to each other and also to other tools from the same area of research.

[2] http://cl-informatik.uibk.ac.at/software/ceta
[3] http://coco.nue.riec.tohoku.ac.jp
[4] http://cops.uibk.ac.at

1.2 Overview

The outline of our thesis is as follows. In Chapter 2 we compile and revisit basic results about conditional term rewriting and related topics that will be indispensable to understand the later chapters. We also give a very brief introduction to interactive theorem proving. The next three chapters present the main confluence methods that we use in our tool. Chapter 3 shortly sums up the result that we may employ certain translations to use confluence methods for standard term rewriting in order to prove confluence of conditional term rewrite systems. Chapter 4 presents our extension and formalization of an earlier orthogonality result by Suzuki et al. Chapter 5 presents our formalization of a modified version of the critical pair criterion for conditional term rewriting originally by Avenhaus and Loría-Sáenz. In Chapter 6 we present some methods to show quasi-decreasingness, a property required by the method of the previous chapter. The methods of Chapters 4 and 5 benefit from being able to ignore certain cases in their confluence analysis. Chapter 7 summarizes various techniques that help us to do just that. Next, Chapter 8 lists some methods to prove conditional term rewrite systems non-confluent. The methods described in Chapter 9 are sometimes useful to make other (non-)confluence methods more applicable. Chapter 10 is basically a system description of the current version of our confluence checker ConCon. Beyond that it also provides plenty of examples on how to use our tool in practice. Chapter 11 describes various experiments using ConCon and other tools. Finally, Chapter 12 wraps up the contents of the earlier chapters and gives some possible future work.

1.3 Chapter Notes

Each of the following chapters concludes with a section called “Chapter Notes”. In these sections we first shortly summarize the contents of the chapter and then give relevant references for the presented results and examples. Sometimes we also highlight related work or point out other interesting facts connected to the topic. Finally, we provide a pointer to the contents of the next chapter.

In this first chapter we have seen a high-level introduction to the topics of automation, formalization, and certification of program analysis.

Most of the proofs in this thesis have been formalized in Isabelle/HOL and are now part of IsaFoR. We choose to only provide proofs of results that we formalized ourselves. Specifically, for the proofs included in Chapters 4, 5, and 7 we want to stress that they are by no means just copies of the earlier results but instead textual descriptions of our actual formalization in Isabelle/HOL. To appreciate the amount of work that went into them we suggest to browse the actual IsaFoR theory files.

Some of the presented techniques have been formalized by others, we choose to include them because they are used by our tool and we want to be self-contained. Notably, the formalization of Theorem 3.1 and Corollary 6.5 is due to Sarah Winkler and René Thiemann. Moreover, some of the methods in Chapter 7 have been formalized by Christian Sternagel, René Thiemann, and Akihisa Yamada (in alphabetical order).

Finally, we want to stress that our work heavily relies on the results that have already been present in IsaFoR and had been included over the years by Christian Sternagel, René Thiemann, and many others.

The following chapter summarizes the basics of (conditional and context-sensitive) term rewriting and other related topics that are needed in order to understand the main results of this thesis.

Chapter 2

Preliminaries

Although we will try to be self-contained some familiarity with the basic notions of (conditional and context-sensitive) term rewriting will be helpful. Furthermore, we will also give some basics on interactive theorem proving in general and on the ***Is**abelle **Fo**rmalization of **R**ewriting* (IsaFoR) in particular.

2.1 Abstract Rewriting

In its simplest form *rewriting* is just replacing one thing with another thing according to some previously established rules. For example if you were to edit a text document on the computer and you found that the name "Gödel" was incorrectly spelled "Godel" you could use the common find-and-replace functionality of the editor together with the rule *"replace o by ö"* to repair it. When we do not care about any properties of the objects we are replacing (in the running example characters in a text) we talk of *abstract rewriting*. Since in abstract rewriting we do not concern ourselves with the characteristics of the entities we are rewriting its properties simply carry over to more involved flavors of rewriting (like term rewriting, context-sensitive rewriting, or conditional term rewriting) which we will encounter later in this chapter. There are some basic but nevertheless important concepts of rewriting which we can already introduce only using abstract rewriting.

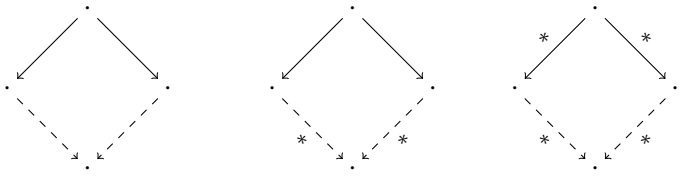
Definition 2.1 (Abstract Rewrite System)**.** An *abstract rewrite system (ARS)* $\mathcal{A}$ is a carrier A together with a binary relation $\to_{\mathcal{A}}$ on A. If $\mathcal{A}$ is clear from context we just write $\to$. Instead of the more common $(a, b) \in \to$ we write $a \to b$ and we call this a *rewrite step*.

Example 2.2. Replacing o's by ö's in a text could be modeled by the ARS $\mathcal{A}$ consisting of carrier $A = \{\text{a,b,c,}\ldots\text{,z,A,B,C,}\ldots\text{,Z,o,ö}\}$ and relation $\to = \{(\text{o}, \text{ö})\}$. We may concisely write $\mathcal{A} = \{\text{o} \to \text{ö}\}$. The only rewrite step $\mathcal{A}$ allows is from o to ö.

To denote the existence of a possibly empty *rewrite sequence* (consisting of zero or more rewrite steps) from a to b we write $a \to^* b$. Here $\to^*$ is the *transitive and reflexive closure* of $\to$. If $a \to^* b$ we say that a *rewrites* to b and we call b a *reduct* of a. In the sequel we will also use $\leftarrow$, $\leftrightarrow$, and $\to^+$ to denote the *inverse*, *symmetric closure*, and *transitive closure* of $\to$, respectively. Moreover, an element $a \in A$ is called *normal form* if there is no $b \in A$ such that $a \to b$. For two relations $\to_\alpha$ and $\to_\beta$ the relations

(a) Commuting diamond property (b) Commutation

Figure 2.1: Some commutation properties.



(a) Diamond property (b) Local confluence (c) Confluence

Figure 2.2: Some confluence properties.

${}^*_\alpha\!\leftarrow \cdot \to^*_\beta$ and $\to^*_\beta \cdot {}^*_\alpha\!\leftarrow$ are called *meetability* and *joinability*. We may abbreviate the relation $\to^* \cdot {}^*\!\leftarrow$ by $\downarrow$. Sometimes we will call a situation $b \,{}^*\!\leftarrow a \to^* c$ a *diverging situation* or a *peak* if it consists of two single steps $b \leftarrow a \to c$. We say that $\to_\alpha$ and $\to_\beta$ *commute* whenever ${}^*_\alpha\!\leftarrow \cdot \to^*_\beta \subseteq \to^*_\beta \cdot {}^*_\alpha\!\leftarrow$ holds. The same property is called confluence, in case α and β coincide.

Definition 2.3 (Confluence, local confluence)**.** An ARS $\mathcal{A}$ is *confluent* if for all $a, b, c \in A$ we have $b \downarrow c$ whenever $b \,{}^*\!\leftarrow a \to^* c$. On the other hand, if we have $b \downarrow c$ whenever $b \leftarrow a \to c$ we call $\mathcal{A}$ *locally confluent*.

In Figures 2.1 and 2.2 we sum up these and some other properties related to commutation and confluence, respectively. The following lemma is a well-known result.

Lemma 2.4. *If $\to_\alpha$ has the diamond property (see Figure 2.2(a)) and the inclusion $\to_\beta \subseteq \to_\alpha \subseteq \to^*_\beta$ holds then $\to_\beta$ is confluent.*

Another property that will be of utmost importance and which can already be defined in this abstract setting is termination.

Definition 2.5 (Termination)**.** An ARS $\mathcal{A}$ is *terminating* if there is no $a \in A$ that admits an infinite rewrite sequence starting from a.

The following famous result by Newman establishes a connection between termination and confluence.

Newman's Lemma. *Every terminating and locally confluent ARS is confluent.*

Sometimes we are interested in the ancestors or descendants of a set of objects with respect to a relation.

Definition 2.6 (Ancestors, descendants)**.** Given a set B we define the set of *ancestors* with respect to $\to$ by $(\to)[B] = \{a \mid a \to b \text{ for some } b \in B\}$. Likewise the set of *descendants* with respect to $\to$ is $[B](\to) = \{a \mid b \to a \text{ for some } b \in B\}$. If B is a singleton we sometimes also just write the sole element instead of B.

In some situations it is helpful to consider one relation relative to another.

Definition 2.7 (Relative rewriting)**.** The relation obtained by considering $\to_\alpha$ *relative to* $\to_\beta$, written $\to_{\alpha/\beta}$, is defined by $\to_\beta^* \cdot \to_\alpha \cdot \to_\beta^*$.

2.2 Term Rewriting

Remember the example from the previous section where we wanted to replace o by ö in order to repair occurrences of the name "Gödel". The ARS from Example 2.2 will replace *all* occurrences of o's by ö's, thereby obfuscating words like "too" to "töö" and so on. What we really wanted to do was to replace the o in all occurrences of "Godel" by ö and leave the other o's alone. In order to properly specify this we need to give the objects we are rewriting more structure. In order to know what our objects should look like we use a so called signature.

Definition 2.8 (Signature)**.** A *signature* is a set $\mathcal{F}$ of *function symbols* with *arity*. For every $f/n \in \mathcal{F}$, f is the function symbol and n is its arity, that is, the number of arguments f is supposed to have. We assume that for two function symbols f/n and g/m if $f = g$ then also $n = m$, that is, we forbid function symbols with the same name but different arity. So in the sequel we will sometimes just write f instead of f/n and whenever we write something like $f = g$ we actually mean $f/n = g/m$.

Sometimes it will be useful to keep some subparts of our objects unfixed. For this we use variables.

Definition 2.9 (Variables)**.** Assume we have a countably infinite *set of variables* $\mathcal{V}$ at our disposal. In the sequel we will use $\mathcal{V}(\cdot)$ to denote the set of variables occurring in a given syntactic object, like a term, a pair of terms, a list of terms, etc. Sometimes we will need the function $\mathsf{var}(t_1, \ldots, t_n)$ that returns the elements of $\mathcal{V}(t_1, \ldots, t_n)$ in an arbitrary but fixed order.

Now a term is a tree-like structure where nodes are labeled by function symbols or variables.

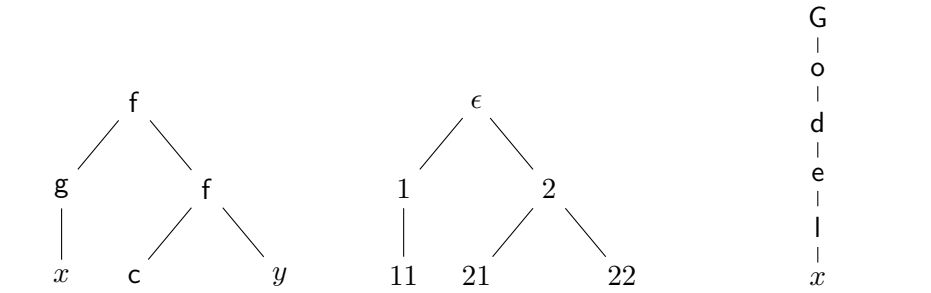
Definition 2.10 (Terms, functional terms, linear terms, ground terms)**.** The set of terms $\mathcal{T}(\mathcal{F}, \mathcal{V})$ over a given signature $\mathcal{F}$ and set of variables $\mathcal{V}$ is defined inductively:

- $x \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ for all variables $x \in \mathcal{V}$, and
- for every n-ary function symbol $f/n \in \mathcal{F}$ and terms $t_1, \ldots, t_n \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ also $f(t_1, \ldots, t_n) \in \mathcal{T}(\mathcal{F}, \mathcal{V})$.

A non-variable term is sometimes also called a *functional term*. If we are not interested in the subterms of a functional term $f(t_1, \ldots, t_n)$ we can just write $f(\ldots)$. A term is called *linear* if it does not contain multiple occurrences of the same variable. Furthermore, we say that a term t is *ground* if $\mathcal{V}(t) = \varnothing$. The *set of ground terms* over $\mathcal{F}$ is denoted by $\mathcal{T}(\mathcal{F})$.

Example 2.11. Consider the signature $\mathcal{F} = \{\mathsf{f}/2, \mathsf{g}/1, \mathsf{c}/0\}$ and the set of variables $\mathcal{V} = \{x, y, z, \ldots\}$. The term $t = \mathsf{f}(\mathsf{g}(x), \mathsf{f}(\mathsf{c}, y))$ is in $\mathcal{T}(\mathcal{F}, \mathcal{V})$. A tree representation of this term is shown in Figure 2.3(a). The term t is linear but not ground.

Example 2.12. Given the signature $\mathcal{G}$, consisting of all upper- and lowercase letters with arity 1 each, and the variable $x \in \mathcal{V}$, the term $\mathsf{G}(\mathsf{o}(\mathsf{d}(\mathsf{e}(\mathsf{l}(x)))))$ is in $\mathcal{T}(\mathcal{G}, \mathcal{V})$. Terms over unary signatures, like $\mathcal{G}$ above, will be called *strings*, and in order to increase readability we drop the parentheses and the single variable and just write Godel. A tree representation of this term is shown in Figure 2.3(c).



(a) The term $\mathsf{f}(\mathsf{g}(x), \mathsf{f}(\mathsf{c}, y))$. (b) Positions in $\mathsf{f}(\mathsf{g}(x), \mathsf{f}(\mathsf{c}, y))$. (c) The term Godel.

Figure 2.3: Tree representation of terms and positions.

Definition 2.13 (Subterm)**.** For two terms s and t in $\mathcal{T}(\mathcal{F}, \mathcal{V})$ we say that t is a *subterm* of s, written $t \trianglelefteq s$ if either $s = t$ or $s = f(s_1, \ldots, s_n)$ and t is a subterm of some s_i. A subterm is proper if $s \neq t$. The *proper subterm* relation is denoted by $\triangleleft$. We can extend any order $\succ$ on terms by the subterm relation in the following way $\succ_{\mathsf{st}} = (\succ \cup \triangleright)^+$.

To reference subterms in a term we use *positions*.

Definition 2.14 (Position)**.** *Positions* in a term are sequences of natural numbers. The empty sequence, called the *root position*, is denoted by ϵ. We denote the subterm of t at position p by $t|_p$. The term we get if we plug in s at position p in t is $t[s]_p$. The set $\mathcal{P}\mathsf{os}(t)$ of all positions in term t is defined recursively:

$$\mathcal{P}\mathsf{os}(t) := \begin{cases} \epsilon & \text{if } t \text{ is a variable} \\ \{ip \mid 1 \leqslant i \leqslant n, p \in \mathcal{P}\mathsf{os}(t|_i)\} & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

Example 2.15. Consider the term $t = \mathsf{f}(\mathsf{g}(x), \mathsf{f}(\mathsf{c}, y))$. The set of positions in t is $\mathcal{P}\mathsf{os}(t) = \{\epsilon, 1, 2, 11, 21, 22\}$ (see Figures 2.3(a) and 2.3(b)). The set of subterms is $\{x, y, \mathsf{c}, \mathsf{g}(x), \mathsf{f}(\mathsf{c}, y), \mathsf{f}(\mathsf{g}(x), \mathsf{f}(\mathsf{c}, y))\}$. We have $t|_1 = \mathsf{g}(x)$, $t|_{21} = \mathsf{c}$, and $t[\mathsf{c}]_2 = \mathsf{f}(\mathsf{g}(x), \mathsf{c})$.

Another possibility to reference subterms of larger terms are *contexts*.

Definition 2.16 (Context, Closure under contexts)**.** *Contexts* are special kinds of terms with a "hole" at one position, where we can plug in other terms. So we first introduce the fresh constant $\Box$ to stand for the *hole*. Now contexts are all terms in the subset of $\mathcal{T}(\mathcal{F} \cup \{\Box\}, \mathcal{V})$ where there is exactly one occurrence of $\Box$. We denote the set of all contexts by $\mathcal{C}(\mathcal{F}, \mathcal{V})$ and we also call $\Box$ the *empty context*. If C is a context, p the position of $\Box$ in the context, and t some term then $C[t]$ denotes the term $C[t]_p$. A binary relation $>$ on terms is *closed under contexts* if $C[s] > C[t]$ for all contexts C and terms s, t where $s > t$.

Example 2.17. Given the context $C = \mathsf{f}(\Box, x)$ and the term $t = \mathsf{g}(y)$ we can combine them to arrive at $C[t] = \mathsf{f}(\mathsf{g}(y), x)$.

Sometimes we are only interested in the symbol at the root position of a term.

Definition 2.18 (Root symbol)**.** Given a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ and a fresh constant $\bot \notin \mathcal{F}$ the function root that returns the *root symbol* of t is defined as follows:

$$\mathsf{root}(t) := \begin{cases} f & \text{if } t = f(t_1, \ldots, t_n) \\ \bot & \text{otherwise} \end{cases}$$

Although the use of $\bot$ in root is unusual, it will be helpful in Section 7.2.

The idea of variables in terms is that we can instantiate them with other terms. In order to use this in rewriting with terms there is still one component missing – substitutions.

Definition 2.19 (Substitution, closure under substitution)**.** A *substitution* is a mapping from variables to terms where only finitely many variables are not mapped to themselves. The *empty substitution* is the identity on $\mathcal{V}$ and written ε. We write $t\sigma$ to denote the result of applying the substitution σ to the term t. Application of a substitution is defined recursively as follows:

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

Sometimes it is useful to represent a substitution by its set of variable bindings $\sigma = \{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$. We call a bijective variable substitution $\pi : \mathcal{V} \to \mathcal{V}$ a *variable renaming* or *(variable) permutation*, and denote its *inverse* by π^-. For two substitutions σ, τ and a set of variables V we write $\sigma = \tau\ [V]$ if $\sigma(x) = \tau(x)$ for all $x \in V$. We write $\sigma\tau$ for the *composition* of the two substitutions σ and τ which is defined to be $(\sigma\tau)(x) = \sigma(x)\tau$, that is, a composition lists substitutions in their order of application. Finally, a substitution σ is called a *ground substitution* if it maps variables to ground terms. A binary relation $>$ on terms is *closed under substitutions* if $s\sigma > t\sigma$ for all substitutions σ and terms s, t where $s > t$.

Example 2.20. Consider the substitution $\sigma = \{x \mapsto y, y \mapsto \mathsf{c}\}$ as well as the term $t = \mathsf{f}(\mathsf{g}(x), \mathsf{f}(\mathsf{c}, y))$. We have $t\sigma = \mathsf{f}(\mathsf{g}(y), \mathsf{f}(\mathsf{c}, \mathsf{c})$ and $\sigma\sigma = \{x \mapsto \mathsf{c}, y \mapsto \mathsf{c}\}$.

With substitutions in place we can now define how to make one term the same as another term – this is called matching.

Definition 2.21 (Matching)**.** A term s *matches* a term t if $t = s\sigma$ for some substitution σ. We say that t is an *instance* of s and conversely that s is a *generalization* of t.

To do term rewriting we need a set of rules that tells us how to rewrite. Similar to ARSs in abstract rewriting we now define term rewrite systems.

Definition 2.22 (Rewrite rule, collapsing rule, TRS)**.** A *rewrite rule* is a pair of terms written $\ell \to r$ where

1. the left-hand side ℓ is not a variable and
2. there are no variables in r that do not already occur in ℓ.

A rule with variable right-hand side is called *collapsing*. We sometimes label rewrite rules like $\rho\colon \ell \to r$. A set of rewrite rules is called a *term rewrite system (TRS)*. A TRS is called *(left-, right-)linear* if all (left-hand, right-hand side) terms are linear. Sometimes we use *extended TRSs* where we only impose the first variable restriction. As usual $\mathcal{R}^{-1}$ denotes the *inverse* of a TRS $\mathcal{R}$. Note that the inverse of a TRS could very well have variable left-hand sides.

Finally, we are ready to rewrite terms. Whenever we find a subterm that matches the left-hand side of a rule in a TRS we can rewrite it to the corresponding right-hand side.

Definition 2.23 (Term rewriting)**.** Given a TRS $\mathcal{R}$ we write $s \to t$ if there exists a rewrite rule $\ell \to r$ in $\mathcal{R}$, a substitution σ and a context C such that $s = C[\ell\sigma]$ and $t = C[r\sigma]$. We call the subterm $\ell\sigma$ the *redex*.

Example 2.24. To come back to our running example, we can specify a TRS that replaces all o's inside "Godel" by ö's as follows: $\mathcal{R} = \{\mathsf{Godel} \to \mathsf{Gödel}\}$. Given for example the term $\mathsf{Godel's\_Incompleteness\_Theorem}$ we can use the single rule in $\mathcal{R}$, substitution $\sigma = \{x \mapsto \mathsf{'s\_Incompleteness\_Theorem}\}$ and the empty context to rewrite it to $\mathsf{Gödel's\_Incompleteness\_Theorem}$. Remember that Godel is shorthand notation for $\mathsf{G}(\mathsf{o}(\mathsf{d}(\mathsf{e}(\mathsf{l}(x)))))$ for some variable x.

Rewriting may also be seen as computation. We can encode numbers as terms and define operations on them. This will become clear from the following example.

Example 2.25. Consider the TRS $\mathcal{R}$

$$x + 0 \to x \qquad\qquad x + \mathsf{s}(y) \to \mathsf{s}(x + y)$$

where in order to increase readability we save parentheses and write + infix, as common in algebra, but + is really just a binary function symbol. Now $\mathcal{R}$ defines addition on natural numbers, in unary notation where 0 encodes 0, $\mathsf{s}(0)$ encodes 1, $\mathsf{s}(\mathsf{s}(0))$ encodes

2, and so on. Let's see addition in action. Assume we want to compute $1 + 2$. The corresponding rewrite sequence looks as follows (to help the reader we underline the redexes):

$$\underline{\mathsf{s}(0) + \mathsf{s}(\mathsf{s}(0))} \to \mathsf{s}(\underline{\mathsf{s}(0) + \mathsf{s}(0)}) \to \mathsf{s}(\mathsf{s}(\underline{\mathsf{s}(0) + 0})) \to \mathsf{s}(\mathsf{s}(\mathsf{s}(0)))$$

In the first rewrite step we use the second rule of $\mathcal{R}$ at the root position. Then, in the second step, we use the same rule but now at position 1. Finally, an application of the first rule of $\mathcal{R}$ at position 2 leads to the term $\mathsf{s}(\mathsf{s}(\mathsf{s}(0)))$, which corresponds to 3, the expected result of the computation $1 + 2$.

In fact term rewriting is a Turing-complete model of computation, that means we can model anything that is computable as a TRS. The terms that cannot be reduced any further with respect to a given TRS (like 0, $\mathsf{s}(0)$, $\mathsf{s}(\mathsf{s}(0))$, and $\mathsf{f}(\mathsf{s}(\mathsf{s}(0)))$ in the example above) are the results of these computations.

Definition 2.26 ($\mathcal{R}$-normal form)**.** Given a TRS $\mathcal{R}$ a term t such that there is no step $t \to_{\mathcal{R}} s$ for any term s is called an *$\mathcal{R}$-normal form* (or just *normal form* if $\mathcal{R}$ is clear from the context).

Sometimes it is convenient to consider substitutions that do not introduce new reducts.

Definition 2.27 ($\mathcal{R}$-normalized substitution)**.** Given a TRS $\mathcal{R}$ a substitution σ where $\sigma(x)$ is an $\mathcal{R}$-normal form for all variables x is called *$\mathcal{R}$-normalized* (or just normalized).

Below we will often use the notion of *rewrite relation* and less frequently the notion of *rewrite order* or *simplification order*.

Definition 2.28 (Rewrite relation, rewrite order, simplification order)**.** A binary relation on terms that is closed under contexts and substitutions is called a *rewrite relation*. Moreover, a proper order on terms that is also a rewrite relation is called a *rewrite order*. Finally, a *simplification order* is a rewrite order with the subterm property.

A property of relations on terms that is sometimes of importance is the following.

Definition 2.29 (Subterm property)**.** A binary relation $>$ on terms is said to have the *subterm property* if $C[t] > t$ for all non-empty contexts C and terms t.

Example 2.30. Of course the proper subterm relation $\rhd$ (see Definition 2.13) has the subterm property (that is where the name comes from after all). In fact $\rhd$ is the smallest proper order that has the subterm property. But $\rhd$ is not a rewrite order because it is not closed under contexts. Now, given a signature $\mathcal{F}$ consider the TRS

$$\mathcal{E}\mathsf{mb}(\mathcal{F}) := \{f(x_1, \ldots, x_n) \to x_i \mid f/n \in \mathcal{F}, 1 \leqslant i \leqslant n\}$$

where $x_1, \ldots, x_n$ are pairwise different variables and $\mathcal{E}\mathsf{mb}(\mathcal{F})$ is short for *embedding rules of signature $\mathcal{F}$*. The relation $\to^+_{\mathcal{E}\mathsf{mb}(\mathcal{F})}$ is the smallest rewrite order that has the subterm property.

Sometimes, for example, when computing critical pairs (see Definition 2.68), we are not just interested in matching one term to another but we want to unify two terms with one substitution.

Definition 2.31 (Unification, unifier, mgu)**.** We say that two terms s and t *unify*, written $s \sim t$, if $s\sigma = t\sigma$ for some substitution σ. In this case we call σ a *unifier*. A substitution σ is as least as general as a substitution τ if there exists a substitution μ such that $\sigma\mu = \tau$. Now a *most general unifier (mgu)* of two terms s and t is as least as general as any other unifier of s and t.

Most of the time the exact variables that occur in a term, rule, etc. are not of interest (since they are only place holders anyway). So we often consider variants.

Definition 2.32 (Variant)**.** For a term t, a rule $\ell \to r$, a substitution σ, etc. and a variable renaming π, the term $t\pi$, the rule $\ell\pi \to r\pi$ (also written $(\ell \to r)\pi$), the substitution $\sigma\pi$ etc. are called *variants* of t, $\ell \to r$, σ, respectively.

Sometimes we are interested in the topmost part of a term that will definitely not change under rewriting with a TRS $\mathcal{R}$. We call this the "cap of the term" or just the *tcap* and we can approximate it by the following function.

Definition 2.33 (tcap)**.**

$$\mathsf{tcap}_{\mathcal{R}}(t) := \begin{cases} u & \text{if } t = f(t_1, \ldots, t_n) \text{ and for all } \ell \to r \in \mathcal{R}.\ u \not\sim \ell \\ y & \text{otherwise} \end{cases}$$

where $u = f(\mathsf{tcap}_{\mathcal{R}}(t_1), \ldots, \mathsf{tcap}_{\mathcal{R}}(t_n))$ and y is a fresh variable.

Example 2.34. Given the TRS $\mathcal{R} = \{\mathsf{a} \to \mathsf{c},\ \mathsf{b} \to \mathsf{d}\}$ computing the term cap yields $\mathsf{tcap}_{\mathcal{R}}(\mathsf{f}(\mathsf{a}, \mathsf{b})) = \mathsf{f}(\mathsf{tcap}_{\mathcal{R}}(\mathsf{a}), \mathsf{tcap}_{\mathcal{R}}(\mathsf{b})) = \mathsf{f}(x, y)$ for two fresh variables x and y.

2.3 Context-Sensitive Rewriting

In general rewrites can take place anywhere in a term in a nondeterministic fashion. To reliably reach normal forms it can sometimes be advantageous to restrict where reductions can take place. One way to do that is to use context-sensitive rewriting, where we only allow reductions at certain positions in a term.

Definition 2.35 (CSRS, replacement map)**.** A *context-sensitive rewrite system (CSRS)* is a TRS over signature $\mathcal{F}$ together with a so called *replacement map* $\mu\colon \mathcal{F} \to 2^{\mathbb{N}}$ that restricts the argument positions of each function symbol in $\mathcal{F}$ at which we are allowed to rewrite. A position p is *active* in a term t if either $p = \epsilon$, or $p = iq$, $t = f(t_1, \ldots, t_n)$, $i \in \mu(f)$, and q is active in t_i. The set of active positions in a term t is denoted by $\mathcal{P}\mathsf{os}_{\mu}(t)$. Given a CSRS $\mathcal{R}$ a term s *μ-rewrites* to a term t, written $s \to_{\mu} t$, if $s \to_{\mathcal{R}} t$ at some position p and $p \in \mathcal{P}\mathsf{os}_{\mu}(s)$.

Example 2.36. Consider the CSRS $\mathcal{R}$

$$\begin{array}{lll} \mathsf{if}(\mathsf{true}, x, y) \to x & \mathsf{and}(\mathsf{true}, x) \to x & 0 + x \to x \\ \mathsf{if}(\mathsf{false}, x, y) \to y & \mathsf{and}(\mathsf{false}, y) \to \mathsf{false} & \mathsf{s}(x) + y \to \mathsf{s}(x + y) \end{array}$$

with replacement map $\mu(\mathsf{if}) = \mu(\mathsf{and}) = \mu(\mathsf{s}) = \mu(+) = \{1\}$. With the help of μ we force the usual behavior of the conditional if as common in most programming languages: the brunches are not touched until the condition is fully evaluated (true or false). Likewise, for the logical connective and we impose a short-circuit evaluation using μ. Let's see this in action:

$$\begin{aligned} \mathsf{if}(\mathsf{and}(\underline{\mathsf{and}(\mathsf{true}, \mathsf{and}(\mathsf{true}, \mathsf{false}))}, x), \mathsf{s}(\mathsf{s}(0)) + y, 0 + \mathsf{s}(0)) &\xrightarrow[\mu]{} \\ \mathsf{if}(\mathsf{and}(\underline{\mathsf{and}(\mathsf{true}, \mathsf{false})}, x), \mathsf{s}(\mathsf{s}(0)) + y, 0 + \mathsf{s}(0)) &\xrightarrow[\mu]{} \\ \mathsf{if}(\underline{\mathsf{and}(\mathsf{false}, x)}, \mathsf{s}(\mathsf{s}(0)) + y, 0 + \mathsf{s}(0)) &\xrightarrow[\mu]{} \\ \mathsf{if}(\mathsf{false}, \mathsf{s}(\mathsf{s}(0)) + y, 0 + \mathsf{s}(0)) &\xrightarrow[\mu]{} \\ 0 + \mathsf{s}(0) & \end{aligned}$$

Because of the replacement map we do not have any choice in this context-sensitive rewrite sequence, only the one shown is possible. On the plus side, whatever (possibly non-terminating) terms we could substitute for x and y, they would never be touched.

A CSRS is called *μ-terminating* if its context-sensitive rewrite relation[1] $\to_\mu$ is terminating. The (proper) subterm relation with respect to replacement map μ, written $\rhd_\mu$, restricts the ordinary subterm relation to active positions. An ordering $>$ on terms $\mathcal{T}(\mathcal{F}, \mathcal{V})$ is called *μ-monotonic* if f is monotonic in its i-th argument whenever $i \in \mu(f)$ for all $f \in \mathcal{F}$, that is,

$$s_i > t_i \implies f(s_1, \ldots, s_{i-1}, s_i, s_{i+1}, \ldots, s_n) > f(s_1, \ldots, s_{i-1}, t_i, s_{i+1}, \ldots, s_n).$$

2.4 Conditional Term Rewriting

Although term rewriting is a Turing-complete model of computation, not all problems can be nicely modeled using it. Sometimes we need more control over rule application. This brings us to conditional term rewriting. Here we allow to equip rewrite rules with conditions, written like $\ell \to r \Leftarrow c$. The intended meaning is that the conditions in c have to hold in order for the rule to be applicable. Over the years many different kinds of conditions (like logical formulas, equations, or inequations) have been proposed. We are interested in conditions that involve the rewrite relation itself. So c will be a (possibly empty) conjunction of equations between terms, written like $s_1 \approx t_1, \ldots, s_k \approx t_k$.

[1] Because it is not closed under contexts $\to_\mu$ is not a rewrite relation in the sense of Definition 2.28.

Definition 2.37 (Conditional rewrite rule, CTRS)**.** A *conditional rewrite rule* is a triple consisting of a term ℓ, the left-hand side, a term r, the right-hand side, and a possibly empty list of equations between terms $s_1 \approx t_1, \ldots, s_k \approx t_k$, written as $\ell \to r \Leftarrow s_1 \approx t_1, \ldots, s_k \approx t_k$. If we are not interested in the details of the conditions we will abbreviate them by c, like $\ell \to r \Leftarrow c$. Sometimes we will use c_i to denote the first i conditions of c and $c_{i,j}$ for the list of conditions $s_i \approx t_i, \ldots, s_j \approx t_j$. Like in the unconditional case we may attach labels to rules $\rho\colon \ell \to r \Leftarrow c$ and left-hand sides of rules are not allowed to be variables but the second variable restriction is more lose as explained below. A *conditional term rewrite system (CTRS)* is a set of conditional rewrite rules. We denote the set of conditional rules (where $k > 0$) in $\mathcal{R}$ by $\mathcal{R}_{\mathsf{c}}$ and the set of unconditional (or standard) rules (where $k = 0$) in $\mathcal{R}$ by $\mathcal{R}_{\mathsf{s}}$. Note that $\mathcal{R} = \mathcal{R}_{\mathsf{c}} \uplus \mathcal{R}_{\mathsf{s}}$.

Conditional rewrite rules and hence also CTRSs are classified according to the distribution of variables among ℓ, r, and c, as follows.

Definition 2.38 (Type of a CTRS, extra variables)**.**

type	requirement
1	$\mathcal{V}(r, c) \subseteq \mathcal{V}(\ell)$
2	$\mathcal{V}(r) \subseteq \mathcal{V}(\ell)$
3	$\mathcal{V}(r) \subseteq \mathcal{V}(\ell, c)$
4	no restrictions

For a conditional rewrite rule $\rho : \ell \to r \Leftarrow c$ the set of *extra variables* is defined as $\mathcal{EV}(\rho) = \mathcal{V}(\rho) - \mathcal{V}(\ell)$. An n-CTRS contains only rules of type n. So a 1-CTRS contains no extra variables, a 2-CTRS may only contain extra variables in the conditions, and a 3-CTRS may also have extra variables in the right-hand sides provided these occur in the corresponding conditional part.

Example 2.39. System $\mathcal{R}_1 = \{\mathsf{f}(x, x) \to \mathsf{a} \Leftarrow \mathsf{g}(x) \approx \mathsf{b}\}$ is a 1-CTRS, whereas system $\mathcal{R}_2 = \{\mathsf{h}(x) \to \mathsf{g}(x) \Leftarrow \mathsf{f}(x, y) \approx \mathsf{b}\}$ is a 2-CTRS, system $\mathcal{R}_3 = \{\mathsf{f}(x) \to \mathsf{g}(y) \Leftarrow x \approx y\}$ is a 3-CTRS, and finally system $\mathcal{R}_4 = \{\mathsf{a} \to \mathsf{f}(x) \Leftarrow y \approx \mathsf{b}\}$ is a 4-CTRS.

Definition 2.40 (Underlying TRS, $\mathcal{R}_{\mathsf{u}}$)**.** The extended TRS obtained from a CTRS $\mathcal{R}$ by dropping the conditional parts of the rewrite rules is denoted by $\mathcal{R}_{\mathsf{u}}$ and called the *underlying TRS* of $\mathcal{R}$. Note that $\to_{\mathcal{R}} \subseteq \to_{\mathcal{R}_u}$.

Example 2.41. The underlying TRS of the CTRS $\mathcal{R}_1$ from Example 2.39 consists of the rewrite rule $\mathsf{f}(x, x) \to \mathsf{a}$.

There are different possibilities for the semantics of an equation $s \approx t$ in the conditions of a conditional rewrite rule. The ones that are most widely used in the literature are

- *semi-equational*, where an equation $s \approx t$ holds if there is a conversion with respect to the rewrite system itself between s and t, that is, $s \leftrightarrow^* t$,
- *join*, where $s \approx t$ holds if s and t are joinable, that is, $s \downarrow t$, and
- *oriented*, where $s \approx t$ holds if t is reachable from s, that is, $s \to^* t$.

A special case of both join and oriented CTRSs are so called *normal* CTRSs. We say that $\mathcal{R}$ is normal if every right-hand side of every condition in every rule is a ground normal form with respect to $\mathcal{R}_{\mathsf{u}}$. In the sequel we will mainly focus on *oriented 3-CTRSs*.

Definition 2.42 (Conditional rewriting)**.** The rewrite relation induced by an oriented CTRS $\mathcal{R}$ is structured into *levels*. For each level i, a TRS $\mathcal{R}_i$ is defined recursively by

$$\mathcal{R}_0 = \varnothing$$
$$\mathcal{R}_{i+1} = \{\ell\sigma \to r\sigma \mid \ell \to r \Leftarrow c \in \mathcal{R} \text{ and } s\sigma \xrightarrow[\mathcal{R}_i]{*} t\sigma \text{ for all } s \approx t \in c\}$$

We write $s \to_{\mathcal{R},n} t$ (or $s \to_n t$ whenever $\mathcal{R}$ is clear from the context) if we have $s \to_{\mathcal{R}_n} t$. The rewrite relation of $\mathcal{R}$ is defined as $\to_{\mathcal{R}} = \bigcup_{i\geqslant 0} \to_{\mathcal{R}_i}$. Furthermore, we write $\sigma, n \vdash c$ whenever $s\sigma \to_n^* t\sigma$ for all $s \approx t$ in c and we say that σ *satisfies* the set of conditions c. If the level n is not important we also write $\sigma \vdash c$. When there is no substitution σ such that $\sigma \vdash c$ we say that the set of conditions c is *infeasible*.

Note that a *conditional rewrite step* $s \to_{\mathcal{R}} t$ employing $\ell \to r \Leftarrow c \in \mathcal{R}$ and substitution σ is only possible if σ satisfies c.

Example 2.43. Consider the following CTRS $\mathcal{R}$ that computes the quicksort algorithm on natural numbers:

$$\begin{aligned}
\rho_0&: && 0 \leqslant x \to \mathsf{true}\\
\rho_1&: && \mathsf{s}(x) \leqslant 0 \to \mathsf{false}\\
\rho_2&: && \mathsf{s}(x) \leqslant \mathsf{s}(y) \to x \leqslant y\\
\rho_3&: && \mathsf{nil} \mathbin{@} x \to x\\
\rho_4&: && (x : xs) \mathbin{@} ys \to x : (xs \mathbin{@} ys)\\
\rho_5&: && \mathsf{split}(x, \mathsf{nil}) \to \langle \mathsf{nil}, \mathsf{nil}\rangle\\
\rho_6&: && \mathsf{split}(x, y : ys) \to \langle xs, y : zs\rangle \Leftarrow \mathsf{split}(x, ys) \approx \langle xs, zs\rangle,\ x \leqslant y \approx \mathsf{true}\\
\rho_7&: && \mathsf{split}(x, y : ys) \to \langle y : xs, zs\rangle \Leftarrow \mathsf{split}(x, ys) \approx \langle xs, zs\rangle,\ x \leqslant y \approx \mathsf{false}\\
\rho_8&: && \mathsf{qsort}(\mathsf{nil}) \to \mathsf{nil}\\
\rho_9&: && \mathsf{qsort}(x : xs) \to \mathsf{qsort}(ys) \mathbin{@} (x : \mathsf{qsort}(zs)) \Leftarrow \mathsf{split}(x, xs) \approx \langle ys, zs\rangle
\end{aligned}$$

Here @ is the infix append operator on lists, : is the usual right-assocciative infix constructor to prepend an element to a list, and $\langle\cdot,\cdot\rangle$ is the pairing constructor.

Assume we want to compute the result of $\mathsf{qsort}(\mathsf{s}(\mathsf{s}(0)) : \mathsf{s}(0) : \mathsf{nil})$ with respect to $\mathcal{R}$. From Definition 2.42 it is clear that the TRS $\mathcal{R}_1$ contains all instances of unconditional rules in $\mathcal{R}$. So the two rewrite sequences

$$\mathsf{s}(\mathsf{s}(0)) \leqslant \mathsf{s}(0) \underset{\rho_2}{\longrightarrow} \mathsf{s}(0) \leqslant 0 \underset{\rho_1}{\longrightarrow} \mathsf{false} \qquad\qquad \mathsf{split}(\mathsf{s}(\mathsf{s}(0)), \mathsf{nil}) \underset{\rho_5}{\longrightarrow} \langle \mathsf{nil}, \mathsf{nil}\rangle$$

only using unconditional rules from $\mathcal{R}$ are in level 1. From these sequences we get a step

$$\mathsf{split}(\mathsf{s}(\mathsf{s}(0)), \mathsf{s}(0) : \mathsf{nil}) \underset{\rho_6}{\longrightarrow} \langle \mathsf{s}(0) : \mathsf{nil}, \mathsf{nil}\rangle$$

in level 2. But then we have the step

$$\mathsf{qsort}(\mathsf{s}(\mathsf{s}(0)) : \mathsf{s}(0) : \mathsf{nil}) \underset{\rho_9}{\longrightarrow} \mathsf{qsort}(\mathsf{s}(0) : \mathsf{nil}) \mathbin{@} (\mathsf{s}(\mathsf{s}(0)) : \mathsf{qsort}(\mathsf{nil}))$$

in level 3. Because

$$\mathsf{split}(\mathsf{s}(0) : \mathsf{nil}) \underset{\rho_5}{\longrightarrow} \langle \mathsf{nil}, \mathsf{nil} \rangle$$

is in level 1 the step

$$\mathsf{qsort}(\mathsf{s}(0) : \mathsf{nil}) \underset{\rho_9}{\longrightarrow} \mathsf{qsort}(\mathsf{nil}) \mathbin{@} (\mathsf{s}(0) : \mathsf{qsort}(\mathsf{nil}))$$

is in level 2. Moreover, we have

$$\mathsf{qsort}(\mathsf{nil}) \underset{\rho_8}{\longrightarrow} \mathsf{nil}$$

in level 1. Putting everything together we get the following conditional rewriting sequence in level 2:

$$\begin{aligned}
\underline{\mathsf{qsort}(\mathsf{s}(\mathsf{s}(0)) : \mathsf{s}(0) : \mathsf{nil})} &\underset{\rho_9}{\longrightarrow} \\
\mathsf{qsort}(\mathsf{s}(0) : \mathsf{nil}) \mathbin{@} (\mathsf{s}(\mathsf{s}(0)) : \underline{\mathsf{qsort}(\mathsf{nil})}) &\underset{\rho_8}{\longrightarrow} \\
\underline{\mathsf{qsort}(\mathsf{s}(0) : \mathsf{nil})} \mathbin{@} (\mathsf{s}(\mathsf{s}(0)) : \mathsf{nil}) &\underset{\rho_9}{\longrightarrow} \\
(\underline{\mathsf{qsort}(\mathsf{nil})} \mathbin{@} (\mathsf{s}(0) : \mathsf{qsort}(\mathsf{nil}))) \mathbin{@} (\mathsf{s}(\mathsf{s}(0)) : \mathsf{nil}) &\underset{\rho_8}{\longrightarrow} \\
(\mathsf{nil} \mathbin{@} (\mathsf{s}(0) : \underline{\mathsf{qsort}(\mathsf{nil})})) \mathbin{@} (\mathsf{s}(\mathsf{s}(0)) : \mathsf{nil}) &\underset{\rho_8}{\longrightarrow} \\
\underline{(\mathsf{nil} \mathbin{@} (\mathsf{s}(0) : \mathsf{nil}))} \mathbin{@} (\mathsf{s}(\mathsf{s}(0)) : \mathsf{nil}) &\underset{\rho_3}{\longrightarrow} \\
\underline{(\mathsf{s}(0) : \mathsf{nil}) \mathbin{@} (\mathsf{s}(\mathsf{s}(0)) : \mathsf{nil})} &\underset{\rho_4}{\longrightarrow} \\
\mathsf{s}(0) : (\underline{\mathsf{nil} \mathbin{@} (\mathsf{s}(\mathsf{s}(0)) : \mathsf{nil})}) &\underset{\rho_3}{\longrightarrow} \\
\mathsf{s}(0) : \mathsf{s}(\mathsf{s}(0)) : \mathsf{nil}
\end{aligned}$$

Most of the time we will not work on the class of all oriented 3-CTRSs but impose further restrictions that are needed to obtain certain properties. A basic restriction that we impose on CTRSs is that their conditions can be evaluated from left to right. This is ensured by the following notion.

Definition 2.44 (Determinism, DCTRS)**.** A conditional rewrite rule

$$\ell \to r \Leftarrow s_1 \approx t_1, \ldots s_k \approx t_k$$

is *deterministic* if $\mathcal{V}(s_i) \subseteq \mathcal{V}(\ell, t_1, \ldots, t_{i-1})$ for all $1 \leqslant i \leqslant k$. An oriented 3-CTRS that only comprises deterministic rules is called a *deterministic CTRS (DCTRS)*.

Example 2.45. Looking back at Example 2.39, $\mathcal{R}_1$ and $\mathcal{R}_3$ are deterministic but $\mathcal{R}_2$ and $\mathcal{R}_4$ are not. The CTRS $\mathcal{R}$ from Example 2.43 is also a DCTRS as is the following two-rule system $\mathcal{R}_5 = \{\mathsf{f}(x, y) \to x \Leftarrow \mathsf{g}(x) \approx z,\ \mathsf{g}(y) \approx z,\ \mathsf{g}(x) \to \mathsf{c} \Leftarrow \mathsf{d} \approx \mathsf{c}\}$.

2.5 Transformations

Since CTRSs are much more involved than TRSs and properties of TRSs are well-understood it is only natural to look for transformations from CTRSs to TRSs in order to be able to apply all the methods that have already been developed for unconditional TRSs to show properties of CTRSs. In its simplest form this means to just forget about the conditions, employing $\mathcal{R}_{\mathsf{u}}$. More sophisticated transformations modify the signature.

Definition 2.46 (Transformation)**.** A *transformation* T is a mapping from CTRSs to TRSs that comes equipped with an *encoding function* $\sharp\colon \mathcal{T}(\mathcal{F},\mathcal{V}) \to \mathcal{T}(\mathcal{F}',\mathcal{V})$ and a partial *decoding function* $\flat\colon \mathcal{T}(\mathcal{F}',\mathcal{V}) \to \mathcal{T}(\mathcal{F},\mathcal{V})$. Here $\mathcal{F}$ is the signature of the CTRS $\mathcal{R}$ under consideration, $\mathcal{F}'$ is the signature of the transformed TRS $\mathsf{T}(\mathcal{R})$, and we require that $\flat(\sharp(t)) = t$ for all terms $t \in \mathcal{T}(\mathcal{F},\mathcal{V})$. If $\flat$ and $\sharp$ are clear from context we will sometimes just write T to denote a transformation.

In case of $\mathcal{R}_{\mathsf{u}}$ both $\sharp$ and $\flat$ are just the identity. An important property of transformations is the following.

Definition 2.47 (Reduction-preservation)**.** Given a transformation $(\mathsf{T},\sharp,\flat)$ we say that T is *reduction-preserving* if whenever $\flat(s) \to^*_{\mathcal{R}} \flat(t)$ then $s \to^*_{\mathsf{T}(\mathcal{R})} t$ for all terms $s,t \in \mathcal{T}(\mathcal{F}',\mathcal{V})$.

The transformation from $\mathcal{R}$ to $\mathcal{R}_{\mathsf{u}}$ obviously has this property. Another important property of transformations is this.

Definition 2.48 (Reduction-reflection)**.** Given a transformation $(\mathsf{T},\sharp,\flat)$ we say that T is *reduction-reflecting* if whenever $\sharp(s) \to^*_{\mathsf{T}(\mathcal{R})} \sharp(t)$ then $s \to^*_{\mathcal{R}} t$ for all terms $s,t \in \mathcal{T}(\mathcal{F},\mathcal{V})$.

It is easy to see that the transformation from $\mathcal{R}$ to $\mathcal{R}_{\mathsf{u}}$ does not have this property (otherwise what would be the point in adding the conditions, anyway?).

We recall two transformations from DCTRSs to TRSs that will be used later. Unravelings split conditional rules into several unconditional rules and the conditions are encoded using new function symbols. Originally they were used to study the correspondence between properties of CTRSs and TRSs as well as modularity of CTRSs. An unraveling simulates the conditional rules from a CTRS $\mathcal{R}$ by a sequence of applications of rules from the TRS $\mathsf{U}(\mathcal{R})$, in effect verifying the conditions from left to right until all the conditions are satisfied and the last rule yielding the original right-hand side may be applied.

Definition 2.49 (Unraveling U)**.** Given a DCTRS $\mathcal{R}$ its *unraveling* $\mathsf{U}(\mathcal{R})$ is defined as follows. For each conditional rule $\rho\colon \ell \to r \Leftarrow s_1 \approx t_1, \ldots, s_k \approx t_k$ in $\mathcal{R}_{\mathsf{c}}$ we introduce k fresh function symbols $U^\rho_1,\ldots,U^\rho_k$ and generate the set of $k+1$ unconditional rules $\mathsf{U}(\rho)$

$$\begin{aligned}
\ell &\to U^\rho_1(s_1,\mathsf{var}(\ell))\\
U^\rho_1(t_1,\mathsf{var}(\ell)) &\to U^\rho_2(s_2,\mathsf{var}(\ell),\mathsf{ev}(t_1))\\
&\vdots\\
U^\rho_k(t_k,\mathsf{var}(\ell),\mathsf{ev}(t_1,\ldots,t_{k-1})) &\to r
\end{aligned}$$

where var and ev denote functions that yield the respective sequences of elements of $\mathcal{V}$ and $\mathcal{EV}$ in some arbitrary but fixed order, and $\mathcal{EV}(t_i) = \mathcal{V}(t_i) \setminus \mathcal{V}(\ell, t_1, \ldots, t_{i-1})$ denotes the *extra variables* of the right-hand side of the ith condition. Finally, the unraveling of the DCTRS $\mathcal{R}$ is $\mathsf{U}(\mathcal{R}) = \mathcal{R}_\mathsf{s} \cup \bigcup_{\rho \in \mathcal{R}_\mathsf{c}} \mathsf{U}(\rho)$.

Example 2.50. Remember the DCTRS $\mathcal{R}_5$ from Example 2.45:

$$\mathsf{f}(x,y) \to x \Leftarrow \mathsf{g}(x) \approx z,\ \mathsf{g}(y) \approx z \qquad\qquad \mathsf{g}(x) \to \mathsf{c} \Leftarrow \mathsf{d} \approx \mathsf{c}$$

It is unraveled into the TRS $\mathsf{U}(\mathcal{R}_5)$ consisting of the following five rules:

$$\begin{aligned} &\mathsf{f}(x,y) \to \mathsf{U}^1_1(\mathsf{g}(x),x,y) \qquad \mathsf{U}^1_1(z,x,y) \to \mathsf{U}^1_2(\mathsf{g}(y),x,y,z) \qquad \mathsf{U}^1_2(z,x,y,z) \to x \\ &\mathsf{g}(x) \to \mathsf{U}^2_1(\mathsf{d},x) \qquad\qquad \mathsf{U}^2_1(\mathsf{c},x) \to \mathsf{c} \end{aligned}$$

The second transformation from DCTRSs to TRSs does not use any additional symbols and it does not aim to simulate rewriting in the DCTRS. Hence its use is limited to show quasi-decreasingness (see Chapter 6).

Definition 2.51 (Transformation V)**.** Every DCTRS $\mathcal{R}$ is mapped to the TRS $\mathsf{V}(\mathcal{R})$ obtained from $\mathcal{R}$ by replacing every conditional rule $\rho\colon \ell \to r \Leftarrow s_1 \approx t_1, \ldots, s_k \approx t_k$ in $\mathcal{R}_\mathsf{c}$ with at most $k+1$ unconditional rules $\mathsf{V}(\rho)$

$$\ell \to s_1\sigma_0 \qquad \cdots \qquad \ell \to s_k\sigma_{k-1} \qquad \ell \to r\sigma_k$$

for the substitutions $\sigma_0, \ldots, \sigma_k$ inductively defined as follows:

$$\sigma_i = \begin{cases} \varepsilon & \text{if } i = 0 \\ \sigma_{i-1} \cup \{x \mapsto s_i\sigma_{i-1} \mid x \in \mathcal{EV}(t_i)\} & \text{if } 1 \leqslant i \leqslant k \end{cases}$$

Finally, the transformed TRS is defined as $\mathsf{V}(\mathcal{R}) = \mathcal{R}_\mathsf{s} \cup \bigcup_{\rho \in \mathcal{R}_\mathsf{c}} \mathsf{V}(\rho)$.

Example 2.52. Again consider the DCTRS $\mathcal{R}_5$ from Example 2.50. Applying transformation V yields the TRS $\mathsf{V}(\mathcal{R}_5)$ consisting of the five rules:

$$\begin{aligned} &\mathsf{f}(x,y) \to \mathsf{g}(x) \qquad \mathsf{f}(x,y) \to \mathsf{g}(y) \qquad \mathsf{f}(x,y) \to x \\ &\mathsf{g}(x) \to \mathsf{d} \qquad\quad \mathsf{g}(x) \to \mathsf{c} \end{aligned}$$

2.6 Tree Automata

When we talk about (possibly infinite) sets of strings (which are really just a special case of terms, see Example 2.12) it is convenient to make use of finite automata. Similarly, if we are interested in (possibly infinite) sets of (full-fledged) terms we may employ *tree automata*, a natural extension of finite automata for strings to terms. Specifically, we will consider bottom-up non-deterministic finite tree automata defined as follows.

Definition 2.53 (Tree automata, language of a tree automaton, regular sets)**.** A bottom-up non-deterministic finite *tree automation (TA)* $\mathcal{A} = \langle \mathcal{F}, Q, Q_f, \Delta \rangle$ consists of four parts:

- a signature $\mathcal{F}$,
- a set of *states* Q disjoint from $\mathcal{F}$,
- a set of *final states* $Q_f \subseteq Q$, and
- a set of *transitions* Δ of the shape
  - $f(q_1, \ldots, q_n) \to q$ with $f/n \in \mathcal{F}$ and $q_1, \ldots, q_n, q \in Q$ or
  - $q \to p$ with $q, p \in Q$.

The *language* of a TA $\mathcal{A}$ is given by the set

$$L(\mathcal{A}) = \{t \in \mathcal{T}(\mathcal{F}) \mid \text{there is a } q \in Q_f \text{ such that } t \xrightarrow[\mathcal{A}]{*} q\}$$

We say that a set of ground terms E is *regular* (or a *regular language*) if there is a TA $\mathcal{A}$ such that $L(\mathcal{A}) = E$.

To represent a TA we usually only need to specify the transitions and mark the final states. The signature and the set of states will be clear from the transitions.

Example 2.54. Consider the TA $\mathcal{A}$ consisting of the following six transitions:

$\mathsf{a} \to 0$	$\mathsf{g}(0) \to 0$	$\mathsf{f}(0,0) \to 0$
$\mathsf{a} \to 1$	$\mathsf{g}(1) \to 2$	$\mathsf{f}(0,2) \to \underline{3}$

Its signature is the set $\mathcal{F} = \{\mathsf{a}/0, \mathsf{g}/1, \mathsf{f}/2\}$, the set of states is $Q = \{0, 1, 2, \underline{3}\}$, and the single final state is $\underline{3}$. The language accepted by $\mathcal{A}$ is the regular set

$$L(\mathcal{A}) = \{\mathsf{f}(t, \mathsf{g}(\mathsf{a})) \mid t \text{ is a ground term over signature } \mathcal{F}\}$$

We have, for example, $\mathsf{f}(\mathsf{a}, \mathsf{g}(\mathsf{a})) \in L(\mathcal{A})$ because of the sequence

$$\mathsf{f}(\mathsf{a}, \mathsf{g}(\mathsf{a})) \underset{\mathcal{A}}{\to} \mathsf{f}(0, \mathsf{g}(\mathsf{a})) \underset{\mathcal{A}}{\to} \mathsf{f}(0, \mathsf{g}(1)) \underset{\mathcal{A}}{\to} \mathsf{f}(0, 2) \underset{\mathcal{A}}{\to} \underline{3}$$

that ends in the final state of $\mathcal{A}$.

When working with TAs we will sometimes need *state substitutions*.

Definition 2.55 (State substitution)**.** A substitution from variables to states is called a *state substitution*.

Example 2.56. Applying the state substitution $\sigma = \{x \mapsto 0, y \mapsto 2\}$ to the term $\mathsf{f}(x, y)$ yields $\mathsf{f}(0, 2)$.

We conclude by summarizing some well-known results which will be used in later chapters. First some closure properties for the languages accepted by TAs.

Theorem 2.57. *The class of regular languages is effectively closed under union, complement, and intersection.* □

It is efficiently decidable if a given ground term is in a given tree language.

Theorem 2.58. *Given a ground term t and a TA $\mathcal{A}$ the question if $t \in L(\mathcal{A})$ is decidable.* □

Finally, we can also efficiently check if the language of a given TA is empty.

Theorem 2.59. *Given a TA $\mathcal{A}$ the question if $L(\mathcal{A}) = \varnothing$ is decidable.* □

2.7 Termination

Termination, that is, the property that the computation defined by (any kind of) rewrite system eventually yields a result, is very important. Although undecidable in general there are many methods that allow to determine termination for many interesting TRSs.

We start by shortly recapitulating some basic notions related to termination.

Definition 2.60 (Reduction order, compatibility)**.** A *reduction order* is a well-founded rewrite order. A TRS $\mathcal{R}$ and a binary relation $>$ on terms are said to be *compatible* if $\ell > r$ for every rewrite rule $\ell \to r \in \mathcal{R}$.

A well-known method to ensure termination is to find a reduction order that is compatible with the TRS.

Theorem 2.61. *A TRS $\mathcal{R}$ is terminating if and only if there is a reduction order $>$ that is compatible with $\mathcal{R}$.* □

One way to get such a reduction order is to interpret the function symbols of the TRS's signature $f \in \mathcal{F}$ as polynomials $f_{\mathcal{I}}$ over the natural numbers in such a way that for all rules $\ell \to r \in \mathcal{R}$ the polynomial representing ℓ is strictly greater (with respect to the standard order on natural numbers) than the one representing r. This method is called *polynomial interpretation*.

Example 2.62. Remember the TRS $\mathcal{R}$ from Example 2.25:

$$x + 0 \to x \qquad\qquad x + \mathsf{s}(y) \to \mathsf{s}(x + y)$$

By using the polynomial interpretation $\mathcal{I}$

$$0_{\mathcal{I}} = 1 \qquad\qquad \mathsf{s}_{\mathcal{I}}(x) = x + 1 \qquad\qquad +_{\mathcal{I}}(x, y) = x + 2y$$

we show termination of $\mathcal{R}$ because we have

$$\begin{aligned} x +_{\mathcal{I}} 0_{\mathcal{I}} &= x + 1 > x \\ x +_{\mathcal{I}} \mathsf{s}_{\mathcal{I}}(y) &= x + 2y + 2 > x + 2y + 1 = \mathsf{s}_{\mathcal{I}}(x +_{\mathcal{I}} y) \end{aligned}$$

for all natural numbers x and y.

Another notion of interest is *simple termination*.

Definition 2.63 (Simple termination)**.** A TRS is said to be *simply terminating* if it is compatible with a simplification order.

A common way to ensure simple termination of a TRS $\mathcal{R}$ (over signature $\mathcal{F}$) is to find a reduction order that is compatible with $\mathcal{R} \cup \mathcal{E}\mathsf{mb}(\mathcal{F})$, where $\mathcal{E}\mathsf{mb}(\mathcal{F})$ is defined like in Example 2.30.

In the case of CTRSs termination of the rewrite relation alone is not enough. We additionally want that the recursive evaluation of the instantiated conditions also terminates. This is called effective termination.

Definition 2.64 (Effective termination)**.** A CTRS $\mathcal{R}$ over signature $\mathcal{F}$ is *effectively terminating* if $\mathcal{R}$ is terminating and for every term $s \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ the set of its $\mathcal{R}$-descendants $[s](\to_{\mathcal{R}}^*)$ is finite and computable.

Unfortunately effective termination is still not enough to guarantee effective computability of the rewrite relation (as suggested by the name *effective* termination). To see why look at the following example.

Example 2.65. Consider the CTRS $\mathcal{R}$ consisting of the single rule

$$\rho\colon \mathsf{a} \to \mathsf{b} \Leftarrow \mathsf{f}(\mathsf{a}) \approx \mathsf{b}$$

This CTRS is effectively terminating because its rewrite relation is empty (see Definition 2.42). In practice, however, the standard approach to evaluate term a is to first try to recursively evaluate $\mathsf{f}(\mathsf{a})$. Since the only available rule is ρ itself we again end up checking its condition *ad infinitum*.

To avoid this kind of behavior we use the stronger property of quasi-decreasingness.

Definition 2.66 (Quasi-decreasingness)**.** A DCTRS $\mathcal{R}$ over signature $\mathcal{F}$ is *quasi-decreasing* if there is a well-founded order $\succ$ on $\mathcal{T}(\mathcal{F}, \mathcal{V})$ such that

- $\succ = \succ_{\mathsf{st}}$,
- $\to_{\mathcal{R}} \subseteq \succ$, and
- for all rules $\ell \to r \Leftarrow s_1 \approx t_1, \ldots, s_k \approx t_k$ in $\mathcal{R}$, all substitutions $\sigma\colon \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$, and all $1 \leqslant i \leqslant k$, if $s_j\sigma \to_{\mathcal{R}}^* t_j\sigma$ for all $1 \leqslant j < i$ then $\ell\sigma \succ s_i\sigma$.

Quasi-decreasingness ensures termination and, for finite CTRSs, effective computability of the rewrite relation.

Theorem 2.67. *Quasi-decreasing DCTRSs are effectively terminating.*

The opposite is not true as witnessed by the CTRS from Example 2.65 (which is an effectively terminating DCTRS that is not quasi-decreasing).

2.8 Confluence

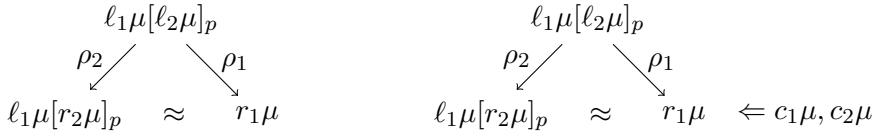
There are numerous ways to show confluence of TRSs. In this section we will recall two basic methods which we will later extend to the conditional case in Chapters 4 and 5.

The first one relies on Newman's Lemma that states that a terminating TRS is confluent if it is locally confluent. To show local confluence, we have to consider all possible peaks $t \leftarrow s \rightarrow u$. Even for finite TRSs the number of peaks may be infinite but it suffices to look at some kind of 'critical' peaks that stem from so called overlaps.

Definition 2.68 (Overlap, critical pair, CP)**.** For a TRS $\mathcal{R}$ over signature $\mathcal{F}$ an *overlap* between two rules $\rho_1 \colon \ell_1 \to r_1$ and $\rho_2 \colon \ell_2 \to r_2$ at position p has the following properties:

1. ρ_1 and ρ_2 are variable-disjoint variants of rules in $\mathcal{R}$,
2. $\ell_1|_p \notin \mathcal{V}$,
3. $\ell_1|_p\mu = \ell_2\mu$ with mgu μ,
4. if $p = \epsilon$ then ρ_1 and ρ_2 are not variants.

If $p = \epsilon$ we call the overlap an *overlay*. The equation $\ell_1\mu[r_2\mu]_p \approx r_1\mu$ is called a *critical pair (CP)* of $\mathcal{R}$ obtained from the above overlap (see Figure 2.4(a)). The set of all critical pairs of $\mathcal{R}$ is denoted by $\mathsf{CP}(\mathcal{R})$.



(a) unconditional case (b) conditional case

Figure 2.4: An overlap and the resulting critical pair.

The following famous result shows why we can concentrate on critical peaks.

Critical Pair Lemma. *A TRS is locally confluent if and only if all its critical pairs are joinable.*

So for a terminating TRS by Newman's Lemma and the Critical Pair Lemma we immediately get the following result:

Corollary 2.69. *A terminating TRS is confluent if and only if all its critical pairs are joinable.*

Note that this means that in the unconditional case confluence is decidable for terminating TRSs.

For the second approach we first need another definition.

Definition 2.70 (Orthogonality)**.** A TRS $\mathcal{R}$ is *orthogonal* if it is left-linear and it has no critical pairs, that is, $\mathsf{CP}(\mathcal{R}) = \varnothing$.

Further, we define *parallel rewriting*, that captures the contraction of pairwise disjoint redexes in one single step.

Definition 2.71 (Parallel rewriting)**.** For a TRS $\mathcal{R}$ we call the relation $\mathrel{+\!\!\!\Rightarrow}_{\mathcal{R}}$ *parallel rewriting* and define it inductively: For two terms s and t we write $s \mathrel{+\!\!\!\Rightarrow}_{\mathcal{R}} t$ if one of the following holds:

- $s = t$,
- $s \to_{\mathcal{R}} t$, or
- $s = f(s_1, \ldots, s_n)$, $t = f(t_1, \ldots, t_n)$, and $s_i \mathrel{+\!\!\!\Rightarrow}_{\mathcal{R}} t_i$ for all $1 \leqslant i \leqslant n$.

This brings us to a very important and well-known result.

Parallel Moves Lemma. *For every orthogonal TRS $\mathcal{R}$ its parallel rewrite relation $\mathrel{+\!\!\!\Rightarrow}_{\mathcal{R}}$ has the diamond property.*

Together with the well-known fact that $\to \subseteq \mathrel{+\!\!\!\Rightarrow} \subseteq \to^*$ and Lemma 2.4 we get:

Corollary 2.72. *An orthogonal TRS is confluent.*

Unlike the previous method this criterion does not depend on the termination of the TRS. Unfortunately, in the conditional case because of the possible extra variables in right-hand sides (and of course the conditions themselves) things are not that easy. But first we have to extend the definition of critical pairs to conditional critical pairs.

Definition 2.73 (Conditional overlap, conditional critical pair, CCP)**.** Given a CTRS $\mathcal{R}$ over signature $\mathcal{F}$ a *conditional overlap* between two rules $\rho_1 \colon \ell_1 \to r_1 \Leftarrow c_1$ and $\rho_2 \colon \ell_2 \to r_2 \Leftarrow c_2$ at position p has the following properties:

1. ρ_1 and ρ_2 are variable-disjoint variants of rules in $\mathcal{R}$,
2. $\ell_1|_p \notin \mathcal{V}$,
3. $\ell_1|_p\mu = \ell_2\mu$ with mgu μ,
4. if $p = \epsilon$ then ρ_1 and ρ_2 are not variants.

A conditional overlap gives rise to a *conditional critical pair (CCP)* (see Figure 2.4(b))

$$\ell_1\mu[r_2\mu]_p \approx r_1\mu \Leftarrow c_1\mu, c_2\mu$$

Example 2.74. Consider the CTRS $\mathcal{R}$ consisting of the two rules

$$\mathsf{f}(\mathsf{g}(x)) \to \mathsf{b} \Leftarrow x \approx \mathsf{a} \qquad\qquad \mathsf{g}(x) \to \mathsf{c} \Leftarrow x \approx \mathsf{c}$$

There is a conditional overlap between the variable-disjoint variants of these rules $\rho_1 \colon \mathsf{f}(\mathsf{g}(y)) \to \mathsf{b} \Leftarrow y \approx \mathsf{a}$ and $\rho_2 \colon \mathsf{g}(z) \to \mathsf{c} \Leftarrow z \approx \mathsf{c}$ at position 1 (hence it is not an overlay) with mgu $\mu = \{y \mapsto z\}$ resulting in the CCP

$$\mathsf{f}(\mathsf{c}) \approx \mathsf{b} \Leftarrow z \approx \mathsf{a},\ z \approx \mathsf{c}$$

Below we will sometimes consider (conditional) overlaps (and the critical pairs induced by them) without the restrictions **2** and **4**. We will call such (C)CPs *improper*.

In general orthogonal CTRSs are not confluent, but we will present an adaptation of the same reasoning to the conditional case in Chapter 4. Likewise the combination of Newman's Lemma and the Critical Pair Lemma does not work out of the box in the conditional case. An adaptation to CTRSs is described in Chapter 5.

Some of the CCPs of a CTRS can be safely ignored because they do not introduce new peaks.

Definition 2.75 (Infeasible CCP)**.** A CCP $s \approx t \Leftarrow c$ is called *infeasible* if c is infeasible.

Example 2.76. Consider the CTRS $\mathcal{R}$ from Example 2.74. Its single CCP

$$\mathsf{f}(\mathsf{c}) \approx \mathsf{b} \Leftarrow z \approx \mathsf{a},\ z \approx \mathsf{c}$$

is infeasible since no term rewrites to both a and c.

Joinability of CCPs differs from the definition in the unconditional case.

Definition 2.77 (Joinability of a CCP)**.** A CCP $s \approx t \Leftarrow c$ is *joinable* if $s\sigma \downarrow_{\mathcal{R}} t\sigma$ for every substitution σ such that $\sigma \vdash c$.

To show joinability of an unconditional CP we just have to find one joining sequence. Showing joinability of a CCP is much harder, because we have to provide joining sequences for all satisfying substitutions.

Example 2.78. The CCP from Example 2.76 above is trivially joinable because its conditions are infeasible.

Example 2.79. Consider the CTRS $\mathcal{R}$ consisting of the four rules

$$\mathsf{f}(x,y) \to \mathsf{f}(\mathsf{g}(\mathsf{s}(x)),y) \Leftarrow \mathsf{c}(\mathsf{g}(x)) \approx \mathsf{c}(\mathsf{a}) \qquad \mathsf{g}(\mathsf{s}(x)) \to x$$
$$\mathsf{f}(x,y) \to \mathsf{f}(x,\mathsf{h}(\mathsf{s}(y))) \Leftarrow \mathsf{c}(\mathsf{h}(y)) \approx \mathsf{c}(\mathsf{a}) \qquad \mathsf{h}(\mathsf{s}(x)) \to x$$

It has one CCP (modulo symmetry)

$$\mathsf{f}(z,\mathsf{h}(\mathsf{s}(v))) \approx \mathsf{f}(\mathsf{g}(\mathsf{s}(z)),v) \Leftarrow \mathsf{c}(\mathsf{g}(z)) \approx \mathsf{c}(\mathsf{a}),\ \mathsf{c}(\mathsf{h}(v)) \approx \mathsf{c}(\mathsf{a})$$

The only substitution that satisfies the conditions is $\sigma = \{z \mapsto \mathsf{s}(\mathsf{a}), v \mapsto \mathsf{s}(\mathsf{a})\}$ and we have the join

$$\mathsf{f}(z,\mathsf{h}(\mathsf{s}(v)))\sigma \underset{\mathcal{R}}{\longrightarrow} \mathsf{f}(\mathsf{g}(\mathsf{s}(\mathsf{s}(\mathsf{a}))),\mathsf{h}(\mathsf{s}(\mathsf{s}(\mathsf{a})))) \underset{\mathcal{R}}{\longleftarrow} \mathsf{f}(\mathsf{g}(\mathsf{s}(z)),v)\sigma$$

using the first conditional rule for the left-hand step and the second conditional rule for the right-hand step.

We sometimes use rules, overlaps, critical pairs, etc. without the adjective "conditional".

2.9 Interactive Theorem Proving

Mathematical proofs may become very long and confusing and also people tend to use their implicit knowledge about a specific area so that a proof based on it may not be clear to another reader. Since long before the rise of the first digital computer Leibniz already had the vision to let machines compute answers to problems by using a simple system of logical inference rules. In the last century digital computers and programming languages became strong enough to put this long sought for goal within our reach. The area of automated theorem proving was born. But computers are not humans and some steps in large proofs still require the ingenuity of the latter. So in *interactive theorem proving* the idea is to not fully automatically try to generate a proof of a statement but by a human-machine collaboration. An interactive theorem prover (also called *proof assistant*) is a software tool that helps a human user to find formal proofs. Typically this involves some sort of interactive proof editor, the interface, in which the human types commands (not unlike programming) and thereby guides the proof search. Under the hood the proof assistant breaks all the commands down to a small kernel of logical inference rules. This kernel is trusted because it is small enough that several experts in the field have verified it. The number of other theorems a new proof might depend on can be mind-boggling and so just searching for the right facts one knows someone has already proved in the past can be very difficult and time consuming. Luckily the automatic support for proof assistants is getting better every year and we are already able to use strong systems, so called *hammers*, that can close tedious but not so difficult gaps in our proofs within seconds. That means the user is able to employ very strong and abstract proof methods while in the background the proof is really constructed from easy, atomic inference steps that are verified by the computer. If the human is able to finish the proof it is formal and mechanized, that is, checkable by the computer without interaction from a human, and thereby as trustworthy as we can get.

Today there are several proof assistants available. The one we used for the presented work is Isabelle. Isabelle is a generic proof assistant that may be instantiated with any one of a selection of specific object logics. We employ *Higher Order Logic* and in the sequel we will refer to the instance of Isabelle we work in as Isabelle/HOL as is customary in the community.

2.10 IsaFoR and CeTA

Before one can formally certify the results of a program on the computer one has to first formalize all of the underlying theory on the computer. In Isabelle/HOL it is possible to generate code for a certifier from the formalization provided all the methods and the corresponding soundness lemmata have been formalized in it first. Work on the ***Is**abelle **Fo**rmalization of **R**ewriting* (IsaFoR) started many years ago. By the time I joined basically everything there is to know about term rewriting was already formalized. My contribution was to extend IsaFoR by results from the conditional term rewriting and tree automata literature. The certifier which is code generated from IsaFoR is

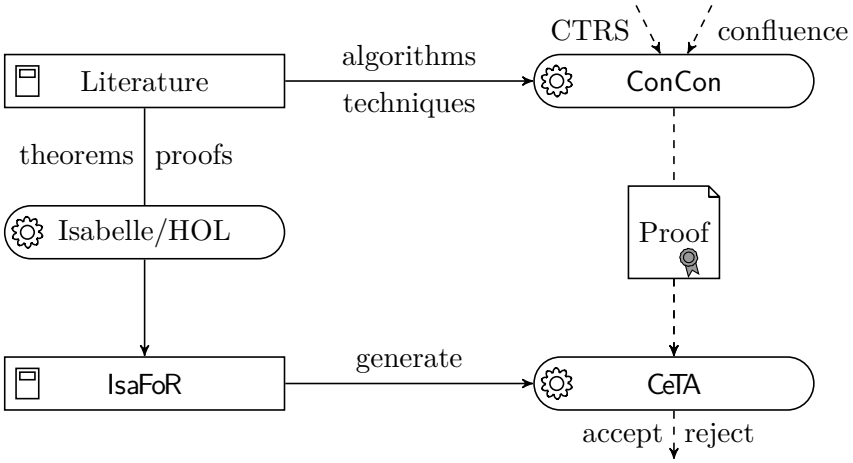


Figure 2.5: Formalization and certification of ConCon.

called CeTA (short for ***Ce**rtified **T**ermination **A**nalysis*; because the initial goal of IsaFoR was certification of termination proofs of TRSs) and is a stand-alone Haskell program. The current version is able to certify all of the methods that we have implemented in our automatic confluence checker for CTRSs – ConCon. A schematic overview of the whole procedure can be seen in Figure 2.5: We first search the literature about results on the properties we want to prove – (non-)confluence, quasi-decreasingness, and infeasibility for conditional term rewriting. Then somewhat orthogonally we on the one hand implement the algorithms and techniques we found in an automatic tool – ConCon – and on the other hand formalize the corresponding theorems and proofs in a proof assistant – Isabelle/HOL. Most of the time, this is not straightforward: Sometimes proofs in the literature contain real errors and we have to fix them before we can proceed. Some other time the authors built their proofs on some implicit assumptions that we have to state explicitly in order for the proof to go through. Almost always there are gaps in the proofs, missing details or tedious technicalities that one may skip on paper and still convince an informed reader about the correctness of the proof but that are needed to convince the computer, that is, Isabelle/HOL. Now the certifier itself is specified and proved correct inside the proof assistant (another round of heavy formalizing) to allow us to automatically code generate it from the formal library – IsaFoR – that is the result of our formalizing work. In order for the certifier to be able to say anything about the output of the automatic tool the output of ConCon has to be adapted. The tool has to provide detailed information about every method it employed in a proof and state this in a format readable by CeTA. Finally, we arrive at an automatic and reliable method to check properties of conditional term rewrite systems: ConCon checks, say confluence, of a given input CTRS automatically. The generated proof is given to CeTA which in turn checks all the tiny steps and accepts the proof if it is correct or rejects it otherwise (also giving a hopefully helpful error message). As a side note: The proof output from ConCon is not just readable by CeTA but it can also be pretty printed in HTML in order to be human readable.

Concerning check functions in our formalization it is worth mentioning that their return type is only "morally" bool. In order to have nice error messages we actually employ a monad. So whenever we need to handle the result of a check function as bool we encapsulate it in a call to *isOK* which results in *False* if there was an error and *True*, otherwise.

We provide the Isabelle/HOL theory files for the formalization that is presented in detail in the subsequent chapters as part of the formal IsaFoR library which depends on the Archive of Formal Proofs (AFP). To be able to browse these files you first need to get the AFP via

```
wget https://www.isa-afp.org/release/afp-current.tar.gz
```

and extract the archive. Then get IsaFoR via

```
hg clone \
  http://cl2-informatik.uibk.ac.at/rewriting/mercurial.cgi/IsaFoR
```

and from inside the `IsaFoR` directory update to tag `16f85a27b856`:

```
hg update -r 16f85a27b856
```

For the remainder, you will need to have Isabelle2016-1 installed. Add the following lines to your `$HOME/.isabelle/Isabelle2016-1/etc/settings`

```
init_component "/path/to/afp/directory/"
init_component "/path/to/isafor/directory"
```

Finally—again from the `IsaFoR` directory—start Isabelle/jEdit in order to browse our formal development. To look at the formalization described in Chapter 5 for example enter the following command:

```
isabelle jedit -l TA thys/Conditional_Rewriting/AL94_Impl.thy
```

This will take some time, even on a (more than) decent machine, the first time around, but will be much faster thereafter.

2.11 Chapter Notes

In this chapter we have seen all the basics of rewriting in general and conditional term rewriting in particular which will be needed in order to follow the discussion in the subsequent chapters.

Much of what is known about term rewriting today may be considered to be folklore. The monograph by Baader and Nipkow [4] as well as the one by Ohlebusch [49] give nice introductions to the field. Our presentation in this chapter is mainly based on these two books as well as an unpublished manuscript by Middeldorp. Together all of the above constitute the most complete collection of important results in term rewriting. The presentation of context-sensitive rewriting is based on Lucas [36]. For a comparison of the many different flavors of conditional term rewriting see [14, p. 586].

Note that in earlier publications (including [25, 27, 48, 66]) reduction-preservation and reduction-reflection have been called *completeness with respect to reductions* and *soundness with respect to reductions*, respectively. This, however, often lead to misunderstandings and we therefore want to try to establish the (in our opinion) more intuitive notions of reduction-reflection and reduction-preservation.

Unravelings were first introduced in [41]. The unraveling U (see Definition 2.49) goes back to [40]. We use the formulation in [49, p. 212]. The transformation V (see Definition 2.51) was first introduced in [3]. There are also other transformations which are known to be complete for certain kinds of CTRSs, including the (optimized) unraveling $\mathsf{U}_{\mathsf{opt}}$ [27, 49], as well as the structure-preserving transformation SR [11], and the complexity-preserving transformation Ξ [33]. An introduction to tree automata may be found in [9]. Newman's Lemma has first been published in [45] and the Critical Pair Lemma is usually attributed to Huet [29]. For more details on Isabelle/HOL see [46], to learn about hammers employed by Isabelle/HOL see [7, 8], and finally, to learn more about CeTA have a look at [65].

Below we give the references for the examples in this chapter, if they are contained in the Cops-database we also provide their Cops-number in parentheses. The CSRS in Example 2.36 is from [36, Example 1]. the first two CTRSs in Example 2.39 are from [58] (`264, 287`), the third is from [23, Example 15] (`390`), and the last is an adaptation of a CTRS from [48, Example 4.18] (`320`). Furthermore, the CTRS in Example 2.43 is from [49, p. 205] (`329`), the one in Example 2.45 is from [48, Example 3.4] (`316`), the one in Example 2.65 is from [23, Example 8] (`386`), the one in Example 2.74 is from [59] (`491`), and the one in Example 2.79 is from [10] (`408`).

The following three chapters present confluence methods for CTRSs. We start in the next chapter by using transformations that allow us to reduce confluence of a conditional system to confluence of a related unconditional system.

Chapter 3

Reduction to the Unconditional Case

In this first chapter on confluence of CTRSs we want to apply a well-tried method, that is, we want to reduce confluence of CTRSs to the well-investigated confluence-analysis of TRSs. To this end we need some kind of transformation that takes a CTRS and produces a corresponding TRS (see Section 2.5). To be able to use such a transformation to show confluence it has to exhibit certain properties. Specifically, it has to be reduction-preserving (see Definition 2.47) so that we can simulate each diverging situation of a CTRS $\mathcal{R}$ in the transformed TRS $\mathsf{T}(\mathcal{R})$. Additionally, it also has to be reduction-reflecting (see Definition 2.48). In this way if we already know that the TRS $\mathsf{T}(\mathcal{R})$ is confluent then we can find a joining sequence for each diverging situation in $\mathsf{T}(\mathcal{R})$ and reduction-reflection also yields a joining sequence in the original CTRS $\mathcal{R}$. So confluence of $\mathsf{T}(\mathcal{R})$ implies confluence of $\mathcal{R}$.

3.1 Formalization

The unraveling U (see Definition 2.49) is reduction-preserving and reduction-reflecting for a certain class of CTRSs.

Theorem 3.1. *For a DCTRS $\mathcal{R}$ if the unraveled TRS $\mathsf{U}(\mathcal{R})$ is left-linear and confluent then $\mathcal{R}$ is confluent.* □

Theorem 3.1 has been formalized in IsaFoR by Winkler and Thiemann. The implementation in ConCon was done by Winkler. Since this is not our achievement we only include a short summary of their result for sake of self-containedness without giving any detailed proofs.

In the uncertified mode of ConCon we employ a different result by Gmeiner et al. which is not formalized in IsaFoR. Their theorem employs the notion of *weak left-linearity*.

Definition 3.2 (Weak left-linearity)**.** A DCTRS $\mathcal{R}$ is said to be *weakly left-linear* if for every rule $\rho\colon \ell \to r \Leftarrow s_1 \approx t_1, \ldots, s_k \approx t_k \in \mathcal{R}$ and all variables $x \in \mathcal{V}(\rho)$ we have that if x occurs more than once in $\ell, t_1, \ldots, t_k$ then x does not occur in $s_1, \ldots, s_k, r$ at all.

The name might suggest that left-linearity implies weak left-linearity but that is not the case. Indeed, weak left-linearity and left-linearity are incomparable as can be seen from the following example:

Example 3.3. The one-rule DCTRS $\mathcal{R}_1 = \{\mathsf{g}(x) \to x \Leftarrow x \approx \mathsf{a},\ \mathsf{b} \approx x\}$ is left-linear but not weakly left-linear. In contrast, the one-rule DCTRS $\mathcal{R}_2 = \{\mathsf{f}(x,x) \to \mathsf{a}\}$ is weakly left-linear but not left-linear.

Theorem 3.4. *A weakly left-linear DCTRS $\mathcal{R}$ is confluent if $\mathsf{U}(\mathcal{R})$ is confluent.*

Theorems 3.4 and 3.1 are not equivalent because there are DCTRSs that are weakly left-linear but their unraveling is not left-linear as witnessed by the following example:

Example 3.5. The DCTRS $\mathcal{R}_3$ consisting of the four rules

$$\mathsf{g}(x) \to \mathsf{b} \qquad \mathsf{a} \to \mathsf{b} \qquad \mathsf{f}(x,x) \to \mathsf{a} \qquad \mathsf{g}(x) \to \mathsf{a} \Leftarrow \mathsf{g}(x) \approx \mathsf{b}$$

is weakly left-linear and its unraveling $\mathsf{U}(\mathcal{R}_3)$

$$\mathsf{g}(x) \to \mathsf{b} \qquad \mathsf{a} \to \mathsf{b} \qquad \mathsf{f}(x,x) \to \mathsf{a} \qquad \mathsf{g}(x) \to \mathsf{U}(\mathsf{g}(x),x) \qquad \mathsf{U}(\mathsf{b}) \to \mathsf{a}$$

is confluent, hence $\mathcal{R}_3$ is confluent by Theorem 3.4. Since $\mathsf{U}(\mathcal{R}_3)$ is not left-linear Theorem 3.1 is not applicable.

On the other hand, left-linearity of the unraveled system implies weak left-linearity of the original system.

Lemma 3.6. *A DCTRS $\mathcal{R}$ is weakly left-linear if $\mathsf{U}(\mathcal{R})$ is left-linear.*

Proof. We concentrate on a single conditional rule $\rho\colon \ell \to r \Leftarrow s_1 \approx t_1,\ \ldots,\ s_k \approx t_k \in \mathcal{R}$. Assume that ρ is not weakly left-linear. That means that there has to exist a variable $x \in \mathcal{V}(\rho)$ such that x occurs more than once in $\ell, t_1, \ldots, t_k$ and $x \in \mathcal{V}(s_1, \ldots, s_k, r)$. The terms $\ell, t_1, \ldots, t_k$ all have to be linear otherwise one of the rules in $\mathsf{U}(\rho)$ (see Definition 2.49) would be non-left-linear. For the same reason x cannot occur more than once in ℓ, t_i for any $1 \leqslant i \leqslant k$. Finally, if x would occur both in t_i and t_j for $1 \leqslant i < j \leqslant k$ the $j+1$-th rule $U^\rho_j(t_j, \mathsf{var}(\ell), \mathsf{ev}(t_1, \ldots, t_{j-1})) \to \cdots$ in $\mathsf{U}(\mathcal{R})$ would be non-left-linear. So if $\mathcal{R}$ is not weakly left-linear then $\mathsf{U}(\mathcal{R})$ is not left-linear. □

That means that Theorem 3.4 subsumes Theorem 3.1.

3.2 Certification

A certificate for Theorem 3.1 has to provide three parts:

- the DCTRS $\mathcal{R}$ for which we want to prove confluence,
- a TRS $\mathcal{R}'$, and finally
- a proof-certificate for the confluence of $\mathcal{R}'$.

Now the certifier first has to check that $\mathcal{R}' = \mathsf{U}(\mathcal{R})$ for some unraveling U. If this is the case and further $\mathcal{R}'$ is left-linear then it remains to check the proof-certificate for $\mathcal{R}'$. To that end it has a variety of techniques already present in IsaFoR at its disposal. On success the certifier concludes that $\mathcal{R}$ is confluent.

3.3 Chapter Notes

We have seen how to use unravelings to reduce confluence of a DCTRS to confluence of a related unconditional TRS.

Theorem 3.1 has first been published by Nishida et al. [48]. Later a generalized version has been formalized and implemented by Winkler and Thiemann [66]. Their formalization does not only work for the unraveling U but for a whole class of so called *standard unravelings*. The notion of weak left-linearity of DCTRSs has been defined in [25]. Theorem 3.4 is due to Gmeiner et al. [27]. It has not been formalized yet.

In our implementation we use the variant of U (called $\mathsf{U}_{\mathsf{conf}}$) sketched in [49, Example 7.2.49] and formalized in [27, Definition 6]. In this variant certain U-symbols originating from different rewrite rules are shared, in order to reduce the number of critical pairs and thereby increasing the chances of obtaining a confluent TRS. Although Winkler and Thiemann showed that $\mathsf{U}_{\mathsf{conf}}$ is not necessarily an optimal choice to analyze confluence (see [66, Example 22]) we obtained the best experimental results using it. The DCTRSs $\mathcal{R}_1$ and $\mathcal{R}_3$ in Examples 3.3 and 3.5 are new, $\mathcal{R}_2$ in Example 3.3 is an adaptation of a CTRS from [58, Example 6].

The formalization of the methods described in this chapter can be found in the following IsaFoR theory files:

```
thys/Conditional_Rewriting/Unraveling.thy
thys/Conditional_Rewriting/Unraveling_Impl.thy
```

The next chapter focuses on how to extend the well-known orthogonality criterion from the unconditional to the conditional case.

Chapter 4

Orthogonality

In the unconditional case the easiest way to show confluence of a TRS is to check for orthogonality. This is a simple syntactic check: the TRS has to be left-linear and no overlaps between left-hand sides of rules are allowed, that means, there are no critical pairs. Then by the Parallel Moves Lemma the parallel rewrite relation of the TRS has the diamond property (see Figure 2.2(a)) and from that confluence immediately follows (see Corollary 2.72). In the present chapter we want to look at a method that may be perceived as an extension of the above result to the conditional case. Due to the presence of conditions and in particular the possibility of extra variables in right-hand sides of rules this result is much more involved for CTRSs.

Example 4.1. Consider the normal 3-CTRS consisting of the three rules

$$\mathsf{a} \to x \Leftarrow \mathsf{g}(x) \approx \mathsf{b} \qquad \mathsf{g}(\mathsf{c}) \to \mathsf{b} \qquad \mathsf{g}(\mathsf{d}) \to \mathsf{b}$$

It is orthogonal but not confluent, since we have the peak $\mathsf{c} \leftarrow \mathsf{a} \to \mathsf{d}$ and the constructor constants c and d are not joinable.

So in general orthogonal CTRSs are *not* confluent. Nevertheless, we can impose further restrictions to arrive at a subclass of orthogonal CTRSs that are confluent. Like in the unconditional case these restrictions are all purely syntactical and hence easy to check.

In this chapter we first present a subclass of orthogonal CTRSs and their properties, known from the literature and then we generalize it in two ways. On the one hand, we relax the syntactic properties and on the other hand, we first extend the notion of orthogonality to allow infeasible conditional critical pairs and then use properties of the proof to enable more powerful methods for proving infeasibility.

4.1 Formalization

Before we look at the orthogonality criterion for the conditional case in detail we have to introduce two properties closely related to confluence. The first of which is *level-commutation*.

Definition 4.2 (Level-commutation)**.** A CTRS $\mathcal{R}$ over signature $\mathcal{F}$ is *level-commuting* if for all levels m, n and terms $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ whenever $s \,{}^{*}_{m}\!\leftarrow \cdot \to^{*}_{n} t$ also $s \to^{*}_{n} \cdot \,{}^{*}_{m}\!\leftarrow t$.

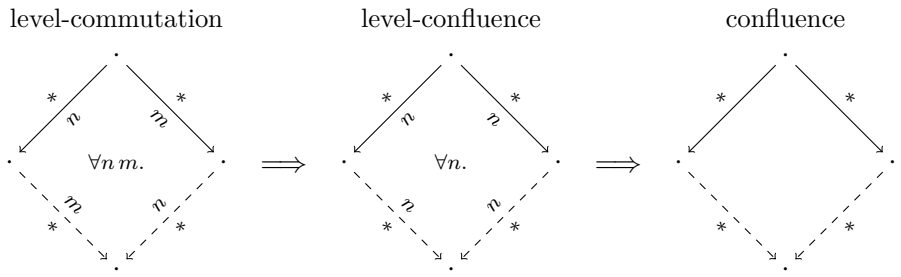


Figure 4.1: Level-commutation, level-confluence, and confluence.

The second property is *level-confluence*.

Definition 4.3 (Level-confluence)**.** A CTRS $\mathcal{R}$ over signature $\mathcal{F}$ is *level-confluent* if for all levels n and terms $s, t, u \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ whenever $s \mathrel{{}^*_n\!\leftarrow} u \rightarrow^*_n t$ also $s \rightarrow^*_n \cdot \mathrel{{}^*_n\!\leftarrow} t$.

The following lemma is depicted in Figure 4.1.

Lemma 4.4. *Level-commutation implies level-confluence which in turn implies confluence.*

Proof. For the first implication just let $n = m$ while for the second we take the maximum of the two levels employed in a peak. □

Unlike orthogonal TRSs, orthogonal CTRSs are not confluent in general, as witnessed by Example 4.1. We have to impose further restrictions on the distribution of variables in the rules such that during matching the substitution for the rewrite step can be built in a deterministic way. To this end we introduce the following notions.

Definition 4.5 (Right-stability, proper orientedness)**.** A conditional rule $\ell \rightarrow r \Leftarrow c$ with k conditions $c = s_1 \approx t_1, \ldots, s_k \approx t_k$ is called

- *right-stable* whenever we have $\mathcal{V}(t_i) \cap \mathcal{V}(\ell, c_{i-1}, s_i) = \varnothing$ and t_i is either a linear constructor term or a ground $\mathcal{R}_u$-normal form, for all $1 \leqslant i \leqslant k$; and
- *properly oriented* if whenever $\mathcal{V}(r) \not\subseteq \mathcal{V}(\ell)$ then $\mathcal{V}(s_i) \subseteq \mathcal{V}(\ell, t_1, \ldots, t_{i-1})$ for all $1 \leqslant i \leqslant k$.

A CTRS consisting solely of right-stable rules is called *right-stable*. Likewise, a CTRS only containing properly oriented rules is called *properly oriented*.

Note that proper orientedness is really just a relaxation of determinism in that we only demand determinism if there are extra variables in right-hand sides of rules and not for all rules (see Definition 2.44). Now the class of CTRSs we are targeting are orthogonal, properly oriented, right-stable, and oriented 3-CTRSs. Remember that in the unconditional case we employed the Parallel Moves Lemma to show confluence of orthogonal systems. So for a CTRS $\mathcal{R}$ we write $s \twoheadrightarrow_n t$ if t can be obtained from s by contracting a set of pairwise disjoint redexes in s using $\mathcal{R}_n$ (see Definition 2.71). Unfortunately, the Parallel Moves Lemma does not hold for our class of CTRSs.

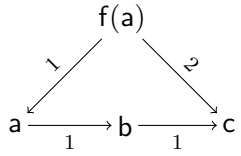


Figure 4.2: For CTRSs parallel rewriting does not have the diamond property.

Example 4.6. Consider the orthogonal, properly oriented, right-stable, and oriented 3-CTRS consisting of the three rules

$$\mathsf{f}(x) \to y \Leftarrow x \approx y \qquad \mathsf{a} \to \mathsf{b} \qquad \mathsf{b} \to \mathsf{c}$$

In the peak depicted in Figure 4.2 no parallel rewrite step from a to c is possible, but we still can rewrite a to the normal form c with a sequence which level is smaller than the level of the step from $\mathsf{f}(\mathsf{a})$ to c. Incorporating these findings in parallel rewriting for CTRSs we arrive at the following notion.

Definition 4.7 (Extended parallel rewriting)**.** First we adopt the convention that the number of holes of a multihole context is denoted by the corresponding lower-case letter, for example, c for C, d for D, e for E etc. Then we say that there is an *extended parallel rewrite step at level n* from s to t, written $s \mathrel{\mathrlap{\to}\!\!\to}_n t$, whenever we have a multihole context C, and sequences of terms $s_1, \ldots, s_c$ and $t_1, \ldots, t_c$, such that $s = C[s_1, \ldots, s_c]$, $t = C[t_1, \ldots, t_c]$, and for all $1 \leqslant i \leqslant k$ we have either

1. $(s_i, t_i) \in \mathcal{R}_n$ (that is, a root step at level n), or
2. $s_i \to^*_{n-1} t_i$.

It is easy to see that $\to_n \subseteq \mathrel{\mathrlap{\to}\!\!\to}_n \subseteq \to^*_n$. We are ready to state the following variation of the Parallel Moves Lemma.

Theorem 4.8. *For orthogonal, properly oriented, right-stable, and oriented 3-CTRSs extended parallel rewriting has the commuting diamond property.* □

Because the commuting diamond property (see Figure 2.1(a)) obviously implies level-commutation the above theorem together with Lemma 4.4 yields the following important result.

Corollary 4.9. *Orthogonal, properly oriented, right-stable, and oriented 3-CTRSs are confluent.* □

We can improve upon the previous result by using a looser notion of proper orientedness.

Definition 4.10 (Extended properly orientedness)**.** A conditional rule $\ell \to r \Leftarrow c$ with k conditions $c = s_1 \approx t_1, \ldots, s_k \approx t_k$ is called *extended properly oriented* when either $\mathcal{V}(r) \subseteq \mathcal{V}(\ell)$ or there is some $0 \leqslant m \leqslant k$ such that $\mathcal{V}(s_i) \subseteq \mathcal{V}(\ell, t_1, \ldots, t_{i-1})$ for

all $1 \leqslant i \leqslant m$ and $\mathcal{V}(r) \cap \mathcal{V}(s_j \approx t_j) \subseteq \mathcal{V}(\ell, t_1, \ldots, t_m)$ for all $m < j \leqslant k$. A CTRS only containing extended properly oriented rules is called an *extended properly oriented* CTRS.

Example 4.11. Consider the oriented 3-CTRS consisting of the single rule

$$\mathsf{g}(x) \to y \Leftarrow x \approx y, z \approx \mathsf{a}$$

It is orthogonal, right-stable, and extended properly oriented but *not* properly oriented, because of the extra variable z in the second condition.

Observe the following property of a conditional rule $\ell \to r \Leftarrow c$ of type 3 with k conditions.

$$\text{for some } 0 \leqslant m \leqslant k.\ \mathcal{V}(r) \subseteq \mathcal{V}(\ell, c_m) \cup (\mathcal{V}(r) \cap \mathcal{V}(c_{m+1,k})) \qquad (\star)$$

which we will use later and which directly follows from $\mathcal{V}(r) \subseteq \mathcal{V}(\ell, c)$. Moreover, we additionally loosen the orthogonality restriction to allow overlaps that are harmless.

Definition 4.12 (Almost Orthogonality modulo Infeasibility)**.** A left-linear CTRS $\mathcal{R}$ is *almost orthogonal (modulo infeasibility)* if each overlap between rules $\ell_1 \to r_1 \Leftarrow c_1$ and $\ell_2 \to r_2 \Leftarrow c_2$ with mgu μ at position p either

1. results from overlapping two variants of the same rule at the root, or
2. is trivial (that is, $p = \epsilon$ and $r_1\mu = r_2\mu$), or
3. is infeasible in the following sense: for arbitrary m and n, whenever levels m and n commute, then it is impossible to satisfy the conditions stemming from the first rule on level m and at the same time the conditions stemming from the second rule on level n. More formally: $\forall m\, n.\, ({}^*_m\!\leftarrow \cdot \to^*_n \subseteq \to^*_n \cdot {}^*_m\!\leftarrow \implies \nexists \sigma.\, \sigma, m \vdash c_1\mu \wedge \sigma, n \vdash c_2\mu)$.

Note that without **2** and **3**, Definition 4.12 corresponds to plain orthogonality. In the following, whenever we talk about *almost orthogonality* we mean Definition 4.12. Observe that the level-commutation assumption of the third alternative in Definition 4.12 allows us to reduce non-meetability to non-joinability. That this is useful in practice is shown by the following example.

Example 4.13 (Non-meetability via tcap)**.** Consider the CTRS consisting of the two rules

$$\mathsf{f}(x) \to \mathsf{a} \Leftarrow x \approx \mathsf{a} \qquad\qquad \mathsf{f}(x) \to \mathsf{b} \Leftarrow x \approx \mathsf{b}$$

This CTRS has the critical pair

$$\mathsf{a} \approx \mathsf{b} \Leftarrow x \approx \mathsf{a}, x \approx \mathsf{b}$$

Since $\mathsf{tcap}(\mathsf{cs}(x, x)) = \mathsf{cs}(y, z) \sim \mathsf{cs}(\mathsf{a}, \mathsf{b})$, where cs is a fresh auxiliary function symbol, we cannot conclude infeasibility via non-reachability analysis using tcap. However, $\mathsf{tcap}(\mathsf{a}) = \mathsf{a} \not\sim \mathsf{b} = \mathsf{tcap}(\mathsf{b})$ shows non-joinability of a and b. By Definition 4.12.**3** this shows non-meetability of a and b and thereby infeasibility of the critical pair.

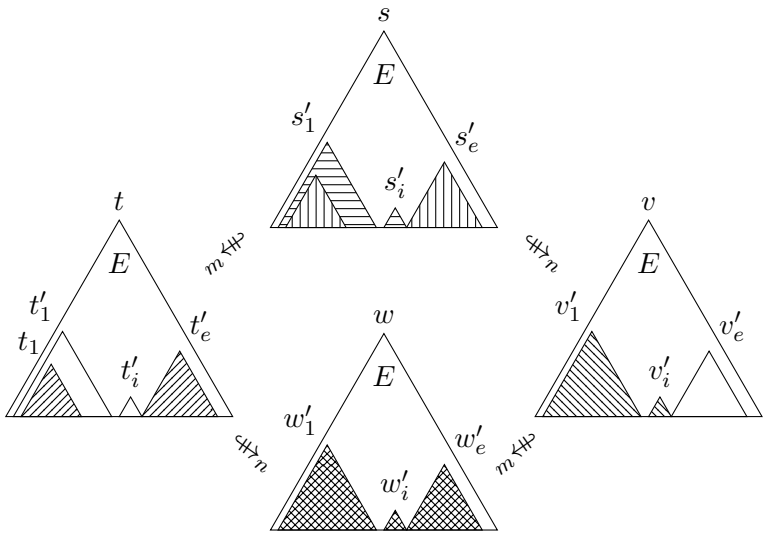


Figure 4.3: Commuting diamond property of extended parallel rewriting.

In general it is beneficial to test for non-meetability via non-joinability of conditions with identical left-hand sides, see also Lemma 7.42. Finally, we are ready to state this new variation of the Parallel Moves Lemma for extended parallel rewriting.

Theorem 4.14. *For almost orthogonal, extended properly oriented, right-stable, and oriented 3-CTRSs extended parallel rewriting has the commuting diamond property.*

Proof. Let $\mathcal{R}$ be a CTRS satisfying all required properties. The commuting diamond property for parallel rewriting states that ${}_m\Leftarrow \cdot \Rightarrow_n \subseteq \Rightarrow_n \cdot {}_m\Leftarrow$ for all m and n. We proceed by complete induction on $m+n$. By induction hypothesis (IH) we may assume the result for all $m'+n' < m+n$. Now consider the peak $t\ {}_m\Leftarrow\ s\ \Rightarrow_n u$. If any of m and n equals 0, we are done (since $\Rightarrow_0$ is the identity relation). Thus we may assume $m = m'+1$ and $n = n'+1$ for some m' and n'. By the definition of extended parallel rewriting, we obtain multihole contexts C and D, and sequences of terms $s_1,\ldots,s_c$, $t_1,\ldots,t_c$, $u_1,\ldots,u_d$, $v_1,\ldots,v_d$, such that $s = C[s_1,\ldots,s_c]$ and $t = C[t_1,\ldots,t_c]$, as well as $s = D[u_1,\ldots,u_d]$ and $u = D[v_1,\ldots,v_d]$; and $(s_i,t_i) \in \mathcal{R}_m$ or $s_i \to^*_{m'} t_i$ for all $1 \leqslant i \leqslant c$, as well as $(u_i,v_i) \in \mathcal{R}_n$ or $u_i \to^*_{n'} v_i$ for all $1 \leqslant i \leqslant d$.

It is relatively easy to define the greatest lower bound $C \sqcap D$ of two contexts C and D by a recursive function (that simultaneously traverses the two contexts in a top-down manner and replaces subcontexts that differ by a hole) and prove that we obtain a lower semilattice. Now we identify the common part E of C and D, employing the semilattice properties of multihole contexts, that is, $E = C \sqcap D$. Then $C = E[C_1,\ldots,C_e]$ and $D = E[D_1,\ldots,D_e]$ for some multihole contexts $C_1,\ldots,C_e$ and $D_1,\ldots,D_e$ such that for each $1 \leqslant i \leqslant e$ we have $C_i = \Box$ or $D_i = \Box$. This also means that there is a sequence of terms $s'_1,\ldots,s'_e$ such that $s = E[s'_1,\ldots,s'_e]$ and for all $1 \leqslant i \leqslant e$, we have $s'_i = C_i[s_{k_i},\ldots,s_{k_i+c_i-1}]$ for some subsequence $s_{k_i},\ldots,s_{k_i+c_i-1}$ of $s_1,\ldots,s_c$ (we denote

similar terms for t, u, and v by t'_i, u'_i, and v'_i, respectively). Moreover, note that by construction $s'_i = u'_i$ for all $1 \leqslant i \leqslant e$. Since extended parallel rewriting is closed under multihole contexts, it suffices to show that for each $1 \leqslant i \leqslant e$ there is a term v such that $t'_i \twoheadrightarrow_n v \;{}_m\!\twoheadleftarrow v'_i$, in order to conclude the proof. This is depicted in Figure 4.3, where w'_i denotes the respective v's. We concentrate on the case $C_i = \square$ (the case $D_i = \square$ is completely symmetric). Moreover, note that when we have $s'_i \rightarrow^*_{m'} t'_i$, the proof concludes by IH (together with some basic properties of the involved relations), and thus we remain with $(s'_i, t'_i) \in \mathcal{R}_m$. At this point we distinguish the following cases:

1. ($D_i = \square$). Also here, the non-root case $u'_i \rightarrow^*_{n'} v'_i$ is covered by the IH. Thus, we may restrict to $(u'_i, v'_i) \in \mathcal{R}_n$, giving rise to a root overlap. Since $\mathcal{R}$ is almost orthogonal, this means that either the resulting conditions are not satisfiable or the resulting terms are the same (in both of these cases we are done), or two variable disjoint variants of the same rule $\ell \rightarrow r \Leftarrow c$ with conditions $c = s_1 \approx t_1, \ldots, s_j \approx t_j$ were involved, that is, $u'_i = \ell\sigma_1 = \ell\sigma_2$ for some substitutions σ_1 and σ_2 that both satisfy all conditions in c. Without extra variables in r, this is the end of the story (since then $r\sigma_1 = r\sigma_2$); but we also want to cover the case where $\mathcal{V}(r) \nsubseteq \mathcal{V}(\ell)$, and thus have to reason why this does not cause any trouble. Together with the fact that $\ell \rightarrow r \Leftarrow c$ is extended properly oriented we obtain a $0 \leqslant k \leqslant j$ such that
   a. $\mathcal{V}(s_i) \subseteq \mathcal{V}(\ell, t_1, \ldots, t_{i-1})$ for all $1 \leqslant i \leqslant k$ and
   b. $\mathcal{V}(r) \cap \mathcal{V}(s_i \approx t_i) \subseteq \mathcal{V}(\ell, t_1, \ldots, t_k)$ for all $k < i \leqslant j$

   by Definition 4.10. Then we prove by an inner induction on $i \leqslant j$ that there is a substitution σ such that
   c. $\sigma(x) = \sigma_1(x) = \sigma_2(x)$ for all x in $\mathcal{V}(\ell)$, and
   d. $\sigma_1(x) \twoheadrightarrow^*_{n'} \sigma(x)$ and $\sigma_2(x) \twoheadrightarrow^*_{m'} \sigma(x)$ for all x in $\mathcal{V}(\ell, c_{\min\{i,k\}}) \cup (\mathcal{V}(r) \cap \mathcal{V}(c_{k+1,i}))$.

   In the base case σ_1 satisfies the requirements. So suppose $i > 0$ and assume by IH that both properties hold for $i - 1$ and some substitution σ. If $i > k$ we are done by **b**. Otherwise $i \leqslant k$. Now consider the condition $s_i \approx t_i$. By **a** we have $\mathcal{V}(s_i) \subseteq \mathcal{V}(\ell, c_{i-1})$. Using the IH for **d** we obtain $s_i\sigma_1 \twoheadrightarrow^*_{n'} s_i\sigma$ and $s_i\sigma_2 \twoheadrightarrow^*_{m'} s_i\sigma$. Moreover $s_i\sigma_1 \twoheadrightarrow^*_{m'} t_i\sigma_1$ and $s_i\sigma_2 \twoheadrightarrow^*_{n'} t_i\sigma_2$ since σ_1 and σ_2 satisfy c, and thus by the outer IH we obtain s' such that $t_i\sigma_1 \twoheadrightarrow^*_{n'} s'$ and $t_i\sigma_2 \twoheadrightarrow^*_{m'} s'$. Recall that by right-stability t_i is either a ground $\mathcal{R}_u$-normal form or a linear constructor term. In the former case $t_i\sigma_1 = t_i\sigma_2 = s'$ and hence σ satisfies **c** and **d**. In the latter case right-stability allows us to combine the restriction of σ_1 to $\mathcal{V}(t_i)$ and the restriction of σ to $\mathcal{V}(\ell, c_{i-1})$ into a substitution satisfying **c** and **d**. This concludes the inner induction. Since $\mathcal{R}$ is an extended properly oriented 3-CTRS, using ($\star$) together with **d** shows $r\sigma_1 \twoheadrightarrow^*_{n'} r\sigma$ and $r\sigma_2 \twoheadrightarrow^*_{m'} r\sigma$. Since $\twoheadrightarrow^*_{n'} \subseteq \twoheadrightarrow_n$ and $\twoheadrightarrow^*_{m'} \subseteq \twoheadrightarrow_m$ we can take $v = r\sigma$ to conclude this case.
2. ($D_i \neq \square$). Then for some $1 \leqslant k \leqslant d$, we have $(u_j, v_j) \in \mathcal{R}_n$ or $u_j \rightarrow^*_{n'} v_j$ for all $k \leqslant j \leqslant k + d_i - 1$, that is, an extended parallel rewrite step of level n from $s'_i = u'_i = D_i[u_{k_i}, \ldots, u_{k_i+d_i-1}]$ to $D_i[v_{k_i}, \ldots, v_{k_i+d_i-1}] = v'_i$. Since $\mathcal{R}$ is almost orthogonal and, by $D_i \neq \square$, root overlaps are excluded, the constituent parts of the extended parallel step from s'_i to v'_i take place exclusively inside the substitution of the root step to the left (which is somewhat obvious but surprisingly hard to

formalize, even more so when having to deal with infeasibility). We again close this case by induction on the number of conditions making use of right-stability of $\mathcal{R}$. □

The same reasoning as before immediately yields the main result of this chapter.

Corollary 4.15. *Almost orthogonal, extended properly oriented, right-stable, and oriented 3-CTRSs are confluent.*

Clearly, applicability of Corollary 4.15 relies on having powerful techniques for proving infeasibility at our disposal. Those are the topic of Chapter 7.

4.2 Certification

Since all the properties of our target CTRSs are syntactical it is straightforward to implement the corresponding check functions. If there are no critical pairs the certificate only contains one element that states that the *almost orthogonal* criterion was used. The syntactic properties are checked by CeTA. More involved proofs contain subproofs of infeasibility for all the CCPs (see Chapter 7).

4.3 Chapter Notes

In this chapter we have seen how the result that orthogonal TRSs are confluent may be extended to a subclass of CTRSs.

While level-commutation is called *shallow confluence* in the literature, we believe that the former is a better, since more descriptive, name. The original result, Theorem 4.8, was published in 1995 by Suzuki et al. [64]. Both the generalization to extended properly oriented CTRSs, as well as the extension to almost orthogonal systems have already been described in the original paper (but without proof). We incorporated both extensions in our formalization and further relaxed the definition of infeasibility to be able to employ commutation in infeasibility proofs for conditional critical pairs (see Definition 4.12.**3**). Note that by dropping **3**, Definition 4.12 reduces to the definition of *almost orthogonality* given by Hanus [28]. The presentation in this chapter is mostly based on our publications [52, 53].

The CTRS in Example 4.1 is similar to an example from [30] (`524`) and the one in Example 4.6 is taken from [64, Example 4.4] (`334`). While the CTRS in Example 4.11 is new, Example 4.13 was already published in [53, Example 3].

The formalization of the methods described in this chapter can be found in the following IsaFoR theory files:

```
thys/Conditional_Rewriting/Level_Confluence.thy
thys/Conditional_Rewriting/Level_Confluence_Impl.thy
```

The next chapter explores a method to show confluence of quasi-decreasing CTRSs provided all of their conditional critical pairs are joinable.

Chapter 5

A Critical Pair Criterion

In the previous chapter we have seen how to adapt the result that orthogonal TRSs are confluent to the conditional case. Here we want to do the same for terminating and locally confluent TRSs. Remember, for terminating TRSs confluence is decidable. We just have to check if all CPs are joinable. This well-known result of unconditional term rewriting directly follows from Newman's Lemma and the Critical Pair Lemma (see Section 2.8). Unfortunately, the same result does not hold in the conditional case. To begin with, the Critical Pair Lemma does not even hold for CTRSs as witnessed by the following example.

Example 5.1. The CTRS $\mathcal{R}$ consisting of the two rules

$$\mathsf{g}(x) \to \mathsf{a} \Leftarrow x \approx \mathsf{g}(x) \qquad\qquad \mathsf{b} \to \mathsf{g}(\mathsf{b})$$

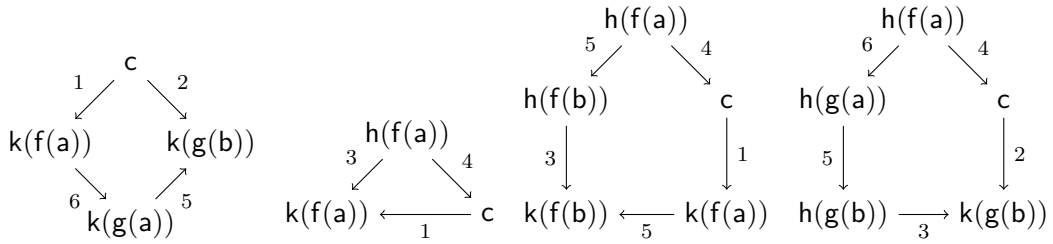
has no CCPs at all but is not (locally) confluent because of the peak $\mathsf{g}(\mathsf{a}) \leftarrow \mathsf{g}(\mathsf{g}(\mathsf{b})) \to \mathsf{a}$, where both steps employ the first rule — the one to the left at position 1 and the one to the right at the root position — and the conditions are satisfied by the second rule. Since $\mathsf{g}(\mathsf{a})$ and a are two different normal forms we have a non-joinable peak and hence a non-confluent system.

Well, the above system is not terminating, maybe terminating CTRSs where all CCPs are joinable are confluent? The next example shows that this is not the case.

Example 5.2. Consider the terminating CTRS $\mathcal{R}$

$$\begin{array}{llllll}
\mathsf{c} \to \mathsf{k}(\mathsf{f}(\mathsf{a})) & (1) & \mathsf{h}(x) \to \mathsf{k}(x) & (3) & \mathsf{a} \to \mathsf{b} & (5) \\
\mathsf{c} \to \mathsf{k}(\mathsf{g}(\mathsf{b})) & (2) & \mathsf{h}(\mathsf{f}(\mathsf{a})) \to \mathsf{c} & (4) & \mathsf{f}(x) \to \mathsf{g}(x) \Leftarrow \mathsf{h}(\mathsf{f}(x)) \approx \mathsf{k}(\mathsf{g}(\mathsf{b})) & (6)
\end{array}$$

There are four CCPs (modulo symmetry) that are all joinable as shown in the diagrams below (where the number of the rule used in a step is attached to the corresponding arrow).



We still have the diverging situation $\mathsf{f}(\mathsf{b}) \leftarrow \mathsf{f}(\mathsf{a}) \rightarrow \mathsf{g}(\mathsf{a}) \rightarrow \mathsf{g}(\mathsf{b})$ where $\mathsf{f}(\mathsf{b})$ and $\mathsf{g}(\mathsf{b})$ are two different normal forms. So $\mathcal{R}$ is not confluent.

Clearly termination is not enough here and so we will employ the stronger notion of quasi-decreasingness which in addition to termination also ensures that the rewrite relation is effectively computable and that there are no infinite computations in the conditions (see Section 2.7). Still, this is not yet sufficient. Because of the possibility of extra variables (see Definition 2.38) in conditional rules there are problems for confluence that can arise from overlaps of a rule with itself at the root position and even from variable-overlaps; two cases that are not dangerous at all when considering unconditional term rewriting. This will become clearer after looking at the following two examples.

Example 5.3. Consider the quasi-decreasing CTRS $\mathcal{R}$ consisting of the following three rules

$$0 + y \rightarrow y \qquad \mathsf{s}(x) + y \rightarrow x + \mathsf{s}(y) \qquad \mathsf{f}(x, y) \rightarrow z \Leftarrow x + y \approx z + v$$

Now look at the (improper) overlap of the last rule with itself. This yields the (improper) CCP $z \approx w \Leftarrow x + y \approx z + v,\ x + y \approx w + u$. Remember that to show joinability of a CCP we have to check whether it is joinable for all satisfying substitutions. Let's see if we can find a satisfying substitution which makes the CCP non-joinable. Well, if we apply the substitution σ that maps x, z, and u to $\mathsf{s}(0)$ and the other variables to 0 to the CCP the result is $\mathsf{s}(0) \approx 0 \Leftarrow \mathsf{s}(0) + 0 \approx \mathsf{s}(0) + 0,\ \mathsf{s}(0) + 0 \approx 0 + \mathsf{s}(0)$. The first condition is trivially satisfied and to satisfy the second one we use the second rule of $\mathcal{R}$. Clearly σ satisfies the CCP. But $\mathsf{s}(0)$ and 0 are two different normal forms and so the CCP is not joinable. Which in turn means that $\mathcal{R}$ is not confluent.

Example 5.4. The quasi-decreasing CTRS $\mathcal{R}$ consists of four rules

$$\mathsf{a} \rightarrow \mathsf{c} \qquad \mathsf{g}(\mathsf{a}) \rightarrow \mathsf{h}(\mathsf{b}) \qquad \mathsf{h}(\mathsf{b}) \rightarrow \mathsf{g}(\mathsf{c}) \qquad \mathsf{f}(x) \rightarrow z \Leftarrow \mathsf{g}(x) \approx \mathsf{h}(z)$$

From the variable-overlap between the first and the last rule we get the (improper) CCP $\mathsf{f}(\mathsf{c}) \approx z \Leftarrow \mathsf{g}(\mathsf{a}) \approx \mathsf{h}(z)$. With $\sigma(z) = \mathsf{b}$ we have found a satisfying substitution that makes the left- and right-hand sides non-joinable.

To counter these problems we have to restrict the placement of extra variables in the conditions severely. To this end we will focus our attention on *strongly deterministic* CTRSs in this chapter.

Definition 5.5 (Strong irreducibility, strong determinism)**.** A term t is called *strongly $\mathcal{R}$-irreducible* if $t\sigma$ is an $\mathcal{R}$-normal form for all $\mathcal{R}$-normalized substitutions σ (see Definition 2.19). Now, a DCTRS (see Definition 2.44) $\mathcal{R}$ is called *strongly deterministic (SDCTRS for short)* if for all rules $\ell \rightarrow r \Leftarrow s_1 \approx t_1, \ldots, s_k \approx t_k$ in $\mathcal{R}$ and $1 \leqslant i \leqslant k$ we have that t_i is strongly $\mathcal{R}$-irreducible.

5.1 Formalization

We are ready to state the main theorem of this chapter.

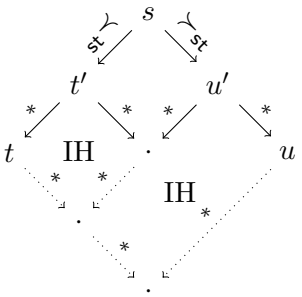
Theorem 5.6. *Let the SDCTRS $\mathcal{R}$ be quasi-decreasing. Then $\mathcal{R}$ is confluent iff all CCPs are joinable.*

Proof. That all CCPs of a CTRS $\mathcal{R}$ (no need for strong determinism or quasi-decreasingness) are joinable if $\mathcal{R}$ is confluent is straightforward. Thus, we concentrate on the other direction: Assume that all critical pairs are joinable. We consider an arbitrary diverging situation $t \,{}^{*}_{\mathcal{R}}\!\leftarrow s \rightarrow^{*}_{\mathcal{R}} u$ and prove $t \downarrow_{\mathcal{R}} u$ by well-founded induction with respect to $\succ_{\mathsf{st}}$. Here $\succ$ is the order from the definition of quasi-decreasingness.

By the induction hypothesis (IH) we have that for all terms t_0, t_1, t_2 such that $s \succ_{\mathsf{st}} t_0$ and $t_1 \,{}^{*}_{\mathcal{R}}\!\leftarrow t_0 \rightarrow^{*}_{\mathcal{R}} t_2$ there exists a join $t_1 \rightarrow^{*}_{\mathcal{R}} \cdot \,{}^{*}_{\mathcal{R}}\!\leftarrow t_2$.

If $s = t$ or $s = u$ then t and u are trivially joinable and we are done. So we may assume that the diverging situation contains at least one step in each direction: $t \,{}^{*}_{\mathcal{R}}\!\leftarrow t' \,{}_{\mathcal{R}}\!\leftarrow s \rightarrow_{\mathcal{R}} u' \rightarrow^{*}_{\mathcal{R}} u$.

Let us show that $t' \downarrow_{\mathcal{R}} u'$ holds. Then $t \downarrow_{\mathcal{R}} u$ follows by two applications of the induction hypothesis, as shown in the following diagram:



Assume that $s = C[\ell_1\sigma_1]_p \rightarrow_{\mathcal{R}} C[r_1\sigma_1]_p = t'$ and $s = D[\ell_2\sigma_2]_q \rightarrow_{\mathcal{R}} D[r_2\sigma_2]_q = u'$ for rules $\rho_1 : \ell_1 \rightarrow r_1 \Leftarrow c_1$ and $\rho_2 : \ell_2 \rightarrow r_2 \Leftarrow c_2$ in $\mathcal{R}$, contexts C and D, positions p and q, and substitutions σ_1 and σ_2 such that $u\sigma_1 \rightarrow^{*}_{\mathcal{R}} v\sigma_1$ for all $u \approx v \in c_1$ and $u\sigma_2 \rightarrow^{*}_{\mathcal{R}} v\sigma_2$ for all $u \approx v \in c_2$. There are three possibilities: either the positions are parallel ($p \parallel q$), or p is above q ($p \leqslant q$), or q is above p ($q \leqslant p$). In the first case $t' \downarrow_{\mathcal{R}} u'$ holds because the two redexes do not interfere. The other two cases are symmetric and we only consider $p \leqslant q$ here. If $s \rhd s|_p = \ell_1\sigma_1$ then $s \succ_{\mathsf{st}} \ell_1\sigma_1$ (by definition of $\succ_{\mathsf{st}}$) and there exists a position r such that $q = pr$ and so we have the diverging situation $r_1\sigma_1 \,{}^{*}_{\mathcal{R}}\!\leftarrow \ell_1\sigma_1 \rightarrow^{*}_{\mathcal{R}} \ell_1\sigma_1[r_2\sigma_2]_r$ which is joinable by the induction hypothesis. But then the diverging situation $t' = s[r_1\sigma_1]_p \,{}^{*}_{\mathcal{R}}\!\leftarrow s[\ell_1\sigma_1]_p \rightarrow^{*}_{\mathcal{R}} s[\ell_1\sigma_1[r_2\sigma_2]_r]_q = u'$ is also joinable (by closure under contexts) and we are done. So we may assume that $p = \epsilon$ and thus $s = \ell_1\sigma_1$. Now, either q is a function position in ℓ_1 or there exists a variable position q' in ℓ_1 such that $q' \leqslant q$. In the first case we either have

1. a conditional critical pair which is joinable by assumption or we have
2. a root-overlap of variants of the same rule. Unlike in the unconditional case this could lead to non-joinability of the ensuing critical pair because of the extra variables

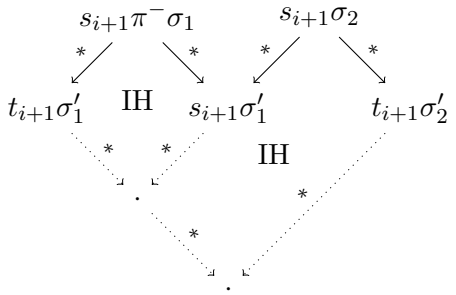
in the right-hand sides of conditional rules. We have $\rho_1\pi = \rho_2$ for some permutation π. Moreover, $s = \ell_1\sigma_1 = \ell_2\sigma_2$ and we have

$$\pi^{-}\sigma_1 = \sigma_2 \ [\mathcal{V}(\ell_2)] \tag{1}$$

We will prove $x\pi^{-}\sigma_1 \downarrow_{\mathcal{R}} x\sigma_2$ for all x in $\mathcal{V}(\rho_2)$. Since $t' = r_1\sigma_1 = r_2\pi^{-}\sigma_1$ and $u' = r_2\sigma_2$ this shows $t' \downarrow_{\mathcal{R}} u'$. Because $\mathcal{R}$ is terminating (by quasi-decreasingness) we may define two normalized substitutions σ'_i such that

$$x\pi^{-}\sigma_1 \xrightarrow[\mathcal{R}]{*} x\sigma'_1 \text{ and } x\sigma_2 \xrightarrow[\mathcal{R}]{*} x\sigma'_2 \text{ for all variables } x. \tag{2}$$

We prove $x\sigma'_1 = x\sigma'_2$ for $x \in \mathcal{EV}(\rho_2)$ by an inner induction on the length k of the conditions $c_2 = s_1 \approx t_1, \ldots, s_k \approx t_k$. If ρ_2 has no conditions this holds vacuously because there are no extra variables. In the step case the inner induction hypothesis is that $x\sigma'_1 = x\sigma'_2$ for $x \in \mathcal{V}(s_1, t_1, \ldots, s_i, t_i) - \mathcal{V}(\ell_2)$ and we have to show that $x\sigma'_1 = x\sigma'_2$ for $x \in \mathcal{V}(s_1, t_1, \ldots, s_{i+1}, t_{i+1}) - \mathcal{V}(\ell_2)$. If $x \in \mathcal{V}(s_1, t_1, \ldots, s_i, t_i, s_{i+1})$ we are done by the inner induction hypothesis and strong determinism of $\mathcal{R}$. So assume $x \in \mathcal{V}(t_{i+1})$. From strong determinism of $\mathcal{R}$, (1), (2), and the inner induction hypothesis we have that $y\sigma'_1 = y\sigma'_2$ for all $y \in \mathcal{V}(s_{i+1})$ and thus $s_{i+1}\sigma'_1 = s_{i+1}\sigma'_2$. With this we can find a join between $t_{i+1}\sigma'_1$ and $t_{i+1}\sigma'_2$ by applying the induction hypothesis twice as shown in the diagram below:



Since t_{i+1} is strongly irreducible and σ'_1 and σ'_2 are normalized, this yields $t_{i+1}\sigma'_1 = t_{i+1}\sigma'_2$ and thus $x\sigma'_1 = x\sigma'_2$.

3. We are left with the case that there is a variable position q' in ℓ_1 such that $q = q'r'$ for some position r'. Let x be the variable $\ell_1|_{q'}$. Then $x\sigma_1|_{r'} = \ell_2\sigma_2$, which implies $x\sigma_1 \to^*_{\mathcal{R}} x\sigma_1[r_2\sigma_2]_{r'}$. Now let τ be the substitution such that $\tau(x) = x\sigma_1[r_2\sigma_2]_{r'}$ and $\tau(y) = \sigma_1(y)$ for all $y \neq x$. Further, let τ' be a normalized substitution where $\tau'(y)$ is some normal form of $\tau(y)$ (which we know must exist) for all y, so $y\tau \to^*_{\mathcal{R}} y\tau'$ for all y. Moreover, note that

$$y\sigma_1 \xrightarrow[\mathcal{R}]{*} y\tau \text{ for all } y. \tag{3}$$

We have $u' = \ell_1\sigma_1[r_2\sigma_2]_q = \ell_1\sigma_1[x\tau]_{q'} \to^*_{\mathcal{R}} \ell_1\tau$, and thus $u' \to^*_{\mathcal{R}} \ell_1\tau'$. From (3) we have $r_1\sigma_1 \to^*_{\mathcal{R}} r_1\tau$ and thus $t' = r_1\sigma_1 \to^*_{\mathcal{R}} r_1\tau'$. Finally, we will show that

$\ell_1\tau' \to_{\mathcal{R}} r_1\tau'$, concluding the proof of $t' \downarrow_{\mathcal{R}} u'$. To this end, let $s_i \approx t_i \in c_1$. By (3) and the definition of τ' we obtain $s_i\sigma_1 \to^*_{\mathcal{R}} t_i\sigma_1 \to^*_{\mathcal{R}} t_i\tau'$ and $s_i\sigma_1 \to^*_{\mathcal{R}} s_i\tau'$. Then $s_i\tau' \downarrow_{\mathcal{R}} t_i\tau'$ by the induction hypothesis and also $s_i\tau' \to^*_{\mathcal{R}} t_i\tau'$, since t_i is strongly irreducible. □

5.2 Certification

There are some complications for employing Theorem 5.6 in practice. Quasi-decreasingness, strong irreducibility, and joinability of CCPs are all undecidable in general. For quasi-decreasingness we fall back to the sufficient criterion that a DCTRS is quasi-decreasing if its unraveling (see Definition 2.49) is terminating (see Chapter 6). A sufficient condition for strong irreducibility is *absolute irreducibility*.

Definition 5.7 (Absolute irreducibility, absolute determinism)**.** A term t is called *absolutely $\mathcal{R}$-irreducible* if none of its non-variable subterms unify with any variable-disjoint variant of left-hand sides of rules in the CTRS $\mathcal{R}$. A DCTRS is called *absolutely deterministic* (or ADCTRS for short) if for each rule all right-hand sides of conditions are absolutely $\mathcal{R}$-irreducible.

The proof of the following lemma is immediate.

Lemma 5.8. *For a term t and a CTRS $\mathcal{R}$:*
- *If t is absolutely $\mathcal{R}$-irreducible, then t is also strongly $\mathcal{R}$-irreducible.*
- *If $\mathcal{R}$ is absolutely deterministic, then $\mathcal{R}$ is also strongly deterministic.* □

We replace joinability of CCPs by infeasibility (see Definition 2.75 and Chapter 7) together with two further criteria which rely on *contextual rewriting*.

Definition 5.9 (Contextual rewriting)**.** Consider a set C of equations between terms which we will call a *context*, but this has nothing to do with the usual use of the word *context* in rewriting which refers to a term with a hole. First we define a function $\bar{\cdot}$ on terms such that $\bar{t}$ is the term t where each variable $x \in \mathcal{V}(C)$ is replaced by a fresh constant $\overline{x}$. (Below we will sometimes call such constants *Skolem constants*.) Moreover, let $\overline{C}$ denote the set C where all variables have been replaced by fresh constants $\overline{x}$. For a CTRS $\mathcal{R}$ we can make a *contextual rewrite step*, denoted by $s \to_{\mathcal{R},C} t$, if we can make a conditional rewrite step with respect to the CTRS $\mathcal{R} \cup \overline{C}$ from $\overline{s}$ to $\overline{t}$.

We formalize soundness of contextual rewriting as follows:

Lemma 5.10. *If $s \to^*_{\mathcal{R},C} t$ then $s\sigma \to^*_{\mathcal{R}} t\sigma$ for all substitutions σ satisfying C.*

Proof. Consider the auxiliary function $[t]_\sigma$, which replaces each Skolem constant $\overline{x}$ in t by $\sigma(x)$, that is, it works like applying a substitution to a term, but to Skolem constants instead of variables. Note that $[\,\bar{t}\,]_\sigma = t\sigma$ whenever $\mathcal{V}(t) \subseteq \mathcal{V}(C)$. Now we show by induction on n that

$$s \to_{\mathcal{R}\cup\overline{C},n} t \text{ implies } [s]_\sigma \to^*_{\mathcal{R},n} [t]_\sigma \qquad (\star)$$

for any σ satisfying C. The base case is trivial. In the inductive step we have a rule $\ell \to r \Leftarrow c \in \mathcal{R} \cup \overline{C}$, a position p, and a substitution τ such that $s|_p = \ell\tau$, $t = s[r\tau]_p$, and $u\tau \to^*_{\mathcal{R}\cup\overline{C},n} v\tau$ for all $u \approx v \in c$. If $\ell \to r \Leftarrow c$ is a rule in $\mathcal{R}$, then we obtain the contextual rewriting sequence $[u\tau]_\sigma \to^*_{\mathcal{R}\cup\overline{C},n} [v\tau]_\sigma$ for all $u \approx v \in c$ by the induction hypothesis. Then we show $s \to^*_{\mathcal{R}\cup\overline{C},n+1} t$ by induction on the context $s[\cdot]_p$. Otherwise, $\ell \to r \Leftarrow c \in \overline{C}$ and thus c is empty, $\ell\tau = \ell$, and $r\tau = r$, since $\overline{C}$ is an unconditional ground TRS. Moreover, there is a rule $\ell' \to r' \in C$ (thus also $\mathcal{V}(\ell', r') \subseteq \mathcal{V}(C)$) such that $\overline{\ell'} = \ell$ and $\overline{r'} = r$. Again, the final result follows by induction on $s[\cdot]_p$.

Assume $s \to_{\mathcal{R},C} t$. Then $\overline{s} \to_{\mathcal{R}\cup\overline{C},n} \overline{t}$ for some level n. Let $\widetilde{t}$ denote the extension of $\overline{t}$ where all variables x in t (not just those in $\mathcal{V}(C)$) are replaced by corresponding fresh constants $\overline{x}$. Note that $\widetilde{t} = \overline{t}\{x \mapsto \overline{x} \mid x \in \mathcal{V}\}$ for every term t. But then we also have $\widetilde{s} \to_{\mathcal{R}\cup\overline{C},n} \widetilde{t}$ since conditional rewriting is closed under substitutions. Note that $[\widetilde{t}]_\sigma = t\sigma$ for all t. Thus taking $\widetilde{s}$ and $\widetilde{t}$ for s and t in $(\star)$ we obtain $s\sigma \to^*_{\mathcal{R},n} t\sigma$. Since we just established the desired property for single contextual rewrite steps it is straightforward to extend it to rewrite sequences. □

Let us illustrate this lemma by an example.

Example 5.11. Remember the CTRS $\mathcal{R}$ from Example 2.79 consisting of the four rules

$$\mathsf{f}(x,y) \to \mathsf{f}(\mathsf{g}(\mathsf{s}(x)),y) \Leftarrow \mathsf{c}(\mathsf{g}(x)) \approx \mathsf{c}(\mathsf{a}) \qquad \mathsf{g}(\mathsf{s}(x)) \to x$$
$$\mathsf{f}(x,y) \to \mathsf{f}(x,\mathsf{h}(\mathsf{s}(y))) \Leftarrow \mathsf{c}(\mathsf{h}(y)) \approx \mathsf{c}(\mathsf{a}) \qquad \mathsf{h}(\mathsf{s}(x)) \to x$$

and the context C containing the two equations

$$\mathsf{c}(\mathsf{g}(x)) \approx \mathsf{c}(\mathsf{a}) \qquad \mathsf{c}(\mathsf{h}(y)) \approx \mathsf{c}(\mathsf{a})$$

We have the sequence

$$\mathsf{f}(x,y) \xrightarrow[\mathcal{R},C]{} \mathsf{f}(\mathsf{g}(\mathsf{s}(x)),y) \xrightarrow[\mathcal{R},C]{} \mathsf{f}(\mathsf{g}(\mathsf{s}(x)),\mathsf{h}(\mathsf{s}(y)))$$

The first step is justified by $\mathsf{f}(\overline{x},\overline{y}) \to_{\mathcal{R}\cup\overline{C}} \mathsf{f}(\mathsf{g}(\mathsf{s}(\overline{x})),\overline{y})$ using the first rule of $\mathcal{R}$ as well as the first equation of $\overline{C}$ to satisfy its condition. For the second step we use $\mathsf{f}(\mathsf{g}(\mathsf{s}(\overline{x})),\overline{y}) \to_{\mathcal{R}\cup\overline{C}} \mathsf{f}(\mathsf{g}(\mathsf{s}(\overline{x})),\mathsf{h}(\mathsf{s}(\overline{y})))$ employing the second rule of $\mathcal{R}$ and the second equation of $\overline{C}$ to satisfy its condition. From Lemma 5.10 we get that for all substitutions σ that satisfy C we have $\mathsf{f}(x,y)\sigma \to \mathsf{f}(\mathsf{g}(\mathsf{s}(x)),\mathsf{h}(\mathsf{s}(y)))\sigma$. In this example the only satisfying substitution is $\sigma = \{x \mapsto \mathsf{s}(\mathsf{a}), y \mapsto \mathsf{s}(\mathsf{a})\}$ employing rules three and four of $\mathcal{R}$.

The above lemma is the key to overcome the undecidability issues of conditional rewriting. For example, for joinability of CCPs the problem is that a single joining sequence (as is usual in certificates for TRSs) does not prove joinability for all satisfying substitutions. However, contextual rewriting has this property.

Now we are able to define the two promised criteria for CCPs that employ contextual rewriting: *context-joinability* and *unfeasibility*.

Definition 5.12 (Unfeasibility, context-joinability)**.** Let $s \approx t \Leftarrow c$ be a CCP induced by an overlap between variable-disjoint variants $\ell_1 \to r_1 \Leftarrow c_1$ and $\ell_2 \to r_2 \Leftarrow c_2$ of rules in $\mathcal{R}$ with mgu μ. We say that the CCP is *unfeasible* if we can find terms u, v, and w such that

1. for all σ that satisfy c we have $\ell_1\mu\sigma \succ u\sigma$,
2. $u \to^*_{\mathcal{R},c} v$,
3. $u \to^*_{\mathcal{R},c} w$, and
4. v and w are both strongly irreducible and $v \not\sim w$.

Moreover, we call the CCP *context-joinable* if there exists some term u such that $s \to^*_{\mathcal{R},c} u$ and $t \to^*_{\mathcal{R},c} u$.

Example 5.13. Consider the CTRS $\mathcal{R}_{\mathsf{last}}$ consisting of the two rules

$$\mathsf{last}(\mathsf{cons}(x,y)) \to x \Leftarrow y \approx \mathsf{nil} \qquad \mathsf{last}(\mathsf{cons}(x,y)) \to \mathsf{last}(y) \Leftarrow y \approx \mathsf{cons}(z,v)$$

$\mathcal{R}_{\mathsf{last}}$ is quasi-decreasing with respect to some well-founded order $\succ$. Moreover, the CTRS has the CCP $x \approx \mathsf{last}(y) \Leftarrow c$ with $c = \{y \approx \mathsf{nil},\ y \approx \mathsf{cons}(z,v)\}$. This CCP is unfeasible because for all satisfying substitutions σ we have $\mathsf{last}(\mathsf{cons}(x,y))\sigma \succ y\sigma$, $y \to^*_{\mathcal{R}_{\mathsf{last}},c} \mathsf{cons}(z,v)$, $y \to^*_{\mathcal{R}_{\mathsf{last}},c} \mathsf{nil}$, and both $\mathsf{cons}(z,v)$ and nil are strongly irreducible and not unifiable.

Now, look at the arbitrary CCP $x \approx \mathsf{min}(\mathsf{nil}) \Leftarrow c$ with $c = \{\mathsf{min}(\mathsf{nil}) \approx x\}$. Since $x \to^*_{\mathcal{R},c} x$ and $\mathsf{min}(\mathsf{nil}) \to^*_{\mathcal{R},c} x$ it is context-joinable (regardless of the actual CTRS $\mathcal{R}$).

Due to Lemma 5.10 above, context-joinability implies joinability of a CCP for arbitrary satisfying substitutions. The rationale for the definition of unfeasibility is a little bit more technical, since it only makes sense inside the proof (by induction) of the theorem below. Basically, unfeasibility is defined in such a way that unfeasible CCPs contradict the confluence of all $\succ$-smaller terms, which we obtain as induction hypothesis.

We are finally ready to state a concrete version of the interesting direction of Theorem 5.6:

Theorem 5.14. *Let the ADCTRS $\mathcal{R}$ be quasi-decreasing. Then $\mathcal{R}$ is confluent if all CCPs are context-joinable, unfeasible, or infeasible.*

Proof. We will denote the well-founded order on terms that we get from quasi-decreasingness by $\succ$. Unfortunately, we cannot directly reuse Theorem 5.6 and its proof, since we need our sufficient criteria in the induction hypothesis. However, the new proof is quite similar. It only differs in case (**1**), where we consider a CCP.

- **a.** If the CCP is context-joinable, we obtain a join with respect to contextual rewriting which we can easily transform into a join with respect to $\mathcal{R}$ by an application of Lemma 5.10 because we have a substitution satisfying the conditions of the CCP.
- **b.** If the CCP is unfeasible, we obtain two diverging contextual rewrite sequences. Again since there is a substitution satisfying the conditions of the CCP we may employ Lemma 5.10 to get two diverging conditional $\mathcal{R}$-rewrite sequences. Because $\ell_1\sigma \succ_{\mathsf{st}} t_0$ we can use the induction hypothesis to get a join between the two end

terms. But from the definition of unfeasibility we also know that the end points are not unifiable (and hence are not the same) and cannot be rewritten (because of strong irreducibility), leading to a contradiction.

c. Finally, if the CCP is infeasible, then there is no substitution that satisfies its conditions, contradicting the fact that we already have such a substitution. □

Have a look at the following example to see Theorem 5.14 in action.

Example 5.15. Consider the quasi-decreasing ADCTRS $\mathcal{R}$ consisting of the following six rules:

$$\mathsf{min}(\mathsf{cons}(x, \mathsf{nil})) \to x \quad (4) \qquad x < 0 \to \mathsf{false} \quad (7)$$
$$\mathsf{min}(\mathsf{cons}(x, xs)) \to x \Leftarrow x < \mathsf{min}(xs) \approx \mathsf{true} \quad (5) \qquad 0 < \mathsf{s}(y) \to \mathsf{true} \quad (8)$$
$$\mathsf{min}(\mathsf{cons}(x, xs)) \to \mathsf{min}(xs) \Leftarrow x < \mathsf{min}(xs) \approx \mathsf{false} \quad (6) \qquad \mathsf{s}(x) < \mathsf{s}(y) \to x < y \quad (9)$$

$\mathcal{R}$ has 6 CCPs, 3 modulo symmetry:

$$x \approx x \Leftarrow x < \mathsf{min}(\mathsf{nil}) \approx \mathsf{true} \quad (1,2)$$
$$x \approx \mathsf{min}(\mathsf{nil}) \Leftarrow x < \mathsf{min}(\mathsf{nil}) \approx \mathsf{false} \quad (1,3)$$
$$x \approx \mathsf{min}(xs) \Leftarrow x < \mathsf{min}(xs) \approx \mathsf{true},\ x < \mathsf{min}(xs) \approx \mathsf{false} \quad (2,3)$$

To conclude confluence of the system it remains to check its CCPs. The first one, (1,2), is trivially context-joinable because the left- and right-hand sides coincide, (1,3) is infeasible since $\mathsf{tcap}(x < \mathsf{min}(\mathsf{nil})) = x < \mathsf{min}(\mathsf{nil})$ and false are not unifiable, and (2,3) is unfeasible because with contextual rewriting we can reach the two non-unifiable normal forms true and false starting from $x < \mathsf{min}(xs)$. Hence, we conclude confluence of $\mathcal{R}$ by Theorem 5.14.

5.3 Certification Challenges

One of the main challenges towards actual certification is typically disregarded on paper: the definition of critical pairs may yield an infinite set of CCPs even for finite CTRSs. This is because we have to consider arbitrary variable-disjoint variants of rules. However, a hypothetical certificate would only contain those CCPs that were obtained from some specific variable-disjoint variants of rules. Now the argument typically goes as follows: *modulo variable renaming there are only finitely many CCPs. Done.*

However, this reasoning is valid only for properties that are either closed under substitution or at least invariant under renaming of variables. For joinability of plain critical pairs—arguably the most investigated case—this is indeed easy. But when it comes to contextual rewriting we spent a considerable amount of work on some results about permutations that were not available in IsaFoR.

To illustrate the issue, consider the abstract specification of the check function *check-CCPs*, such that *isOK* (*check-CCPs* $\mathcal{R}$) implies that each of the CCPs of $\mathcal{R}$ is either unfeasible, context-joinable, or infeasible. To this end we work modulo the assumption

that we already have sound check functions for the latter three properties, which is nicely supported by Isabelle's locale mechanism:

locale `al94-spec` =
  fixes v_x **and** v_y
    and *check-context-joinable*
    and *check-infeasible*
    and *check-unfeasible*
  assumes v_x and v_y are injective
    and $\mathrm{ran}(v_x) \cap \mathrm{ran}(v_y) = \varnothing$
    and `isOK` (*check-context-joinable* $\mathcal{R}$ s t C) $\Longrightarrow \exists u.\, s \to^*_{\mathcal{R},C} u \wedge t \to^*_{\mathcal{R},C} u$)
      ...

For technical reasons, our formalization uses two locales (`al94-ops`, `al94-spec`) here. We just list the required properties of the renaming functions v_x and v_y and the soundness assumption for *check-context-joinable*.

Now what would a certificate contain and how would we have to check it? Amongst other things, the certificate would contain a finite set of CCPs $\mathcal{C}'$ that were computed by some automated tool. Internally, our certifier computes its own finite set of CCPs $\mathcal{C}$ where variable-disjoint variants of rules are created by fixed injective variable renaming functions v_x and v_y, whose ranges are guaranteed to be disjoint. The former prefixes the character "x" and the latter the character "y" to all variable names, hence the names. At this point we have to check that for each CCP in $\mathcal{C}$ there is one in $\mathcal{C}'$ that is its variant, which is not too difficult. More importantly, we have to prove that whenever some desired property P, say context-joinability, holds for any CCP, then P also holds for all of its variants (including the one that is part of $\mathcal{C}$).

To this end, assume that we have a CCP resulting from a critical overlap of the two rules $\ell_1 \to r_1 \Leftarrow c_1$ and $\ell_2 \to r_2 \Leftarrow c_2$ at position p with mgu μ. This means that there exist permutations π_1 and π_2 such that $(\ell_1 \to r_1 \Leftarrow c_1)\pi_1$ and $(\ell_2 \to r_2 \Leftarrow c_2)\pi_2$ are both in $\mathcal{R}$. In our certifier, mgus are computed by the function $\mathrm{mgu}(s,t)$ which either results in `None`, if $s \not\sim t$, or in `Some` μ such that μ is an mgu of s and t, otherwise. Moreover, variable-disjointness of rules is ensured by v_x and v_y, so that we actually call $\mathrm{mgu}(\ell_1|_p\pi_1 v_x, \ell_2\pi_2 v_x)$ for computing a concrete CCP corresponding to the one we assumed above. Thus, we need to show that $\mathrm{mgu}(\ell_1|_p, \ell_2) =$ `Some` μ also implies that $\mathrm{mgu}(\ell_1|_p\pi_1 v_x, \ell_2\pi_2 y_v) =$ `Some` μ' for some mgu μ'. Moreover, we are interested in the relationship between μ and μ' with respect to the variables in both rules. Previously—for an earlier formalization of infeasibility [52]—IsaFoR only contained a result that related both unifiers modulo some arbitrary substitution (that is, not necessarily a renaming).

Unfortunately, contextual rewriting is not closed under arbitrary substitutions. Nevertheless, contextual rewriting is closed under permutations, provided the permutation is also applied to C.

Lemma 5.16. *For every permutation π we have that $s\pi \to^*_{R,C\pi} t\pi$ iff $s \to^*_{R,C} t$.* □

It remains to show that μ and μ' differ basically only by a renaming (at least on the variables of our two rules), which is covered by the following lemma.

Lemma 5.17. *Let* $\mathrm{mgu}(s,t) = \texttt{Some}\ \mu$ *and* $\mathcal{V}(s,t) \subseteq S \cup T$ *for two finite sets of variables* S *and* T *with* $S \cap T = \varnothing$*. Then, there exist a substitution* μ' *and a permutation* π *such that for arbitrary permutations* π_1 *and* π_2*:* $\mathrm{mgu}(s\pi_1 v_x, t\pi_2 v_y) = \texttt{Some}\ \mu'$*,* $\mu = \pi_1\mu' v_x\pi\ [S]$*, and* $\mu = \pi_2\mu' v_y\pi\ [T]$*.*

Proof. Let $h(x) = xv_x\pi_1$ if $x \in S$ and $h(x) = xv_y\pi_2$, otherwise. Then, since h is bijective between $S \cup T$ and $h(S \cup T)$ we can obtain a permutation π for which $\pi = h\ [S \cup T]$. We define $\mu' = \pi^-\mu$ and abbreviate $s\pi_1 v_x$ and $t\pi_2 v_y$ to s' and t', respectively. Note that $s' = s\pi$ and $t' = t\pi$. Since μ is an mgu of s and t we have $s\mu = t\mu$, which further implies $s'\mu' = t'\mu'$. But then μ' is a unifier of s' and t' and thus there exists some μ'' for which $\mathrm{mgu}(s',t') = \texttt{Some}\ \mu''$ and $s'\mu'' = t'\mu''$.

We now show that μ' is also most general. Assume $s'\tau = t'\tau$ for some τ. Then $s\pi\tau = t\pi\tau$ and thus there exists some δ such that $\pi\tau = \mu\delta$ (since μ is most general). But then $\pi^-\pi\tau = \pi^-\mu\delta$ and thus $\tau = \mu'\delta$. Hence, μ' is most general.

Since μ'' is most general too, it only differs by a renaming, say π', from μ', that is, $\mu'' = \pi'\mu'$. This yields $\mu = \pi_1\mu'' v_x\pi'^-\ [S]$ and $\mu = \pi_2\mu'' v_y\pi'^-[T]$, and thus concludes the proof. □

5.4 Check Functions

Before we can actually certify the output of CTRS confluence tools with CeTA, we have to provide an executable check function for each property that is required to apply Theorem 5.14 and prove its soundness. For the check functions for quasi-decreasingness and infeasibility see Chapters 6 and 7, respectively. It remains to provide new check functions for absolute irreducibility, absolute determinism, contextual rewrite sequences, context-joinability, and unfeasibility together with their corresponding soundness proofs. For absolute irreducibility we provide the check function *check-airr*, employing existing machinery from IsaFoR for renaming and unification, and prove:

Lemma 5.18. `isOK` (*check-airr* $\mathcal{R}$ t) *iff the term* t *is absolutely* $\mathcal{R}$*-irreducible.* □

This, in turn, is used to define the check function *check-adtrs* and the accompanying lemma for ADCTRSs.

Lemma 5.19. `isOK` (*check-adtrs* $\mathcal{R}$) *iff* $\mathcal{R}$ *is an ADCTRS.* □

Concerning contextual rewriting, we provide the check function *check-csteps* for conditional rewrite sequences together with the following lemma:

Lemma 5.20. *Given a CTRS* $\mathcal{R}$*, a set of conditions* C*, two terms* s *and* t*, and a list of conditional rewrite proofs* ps*, we have that* `isOK` (*check-csteps* $(\mathcal{R} \cup \overline{C})\ \overline{s}\ \overline{t}\ \overline{ps}$) *implies* $s \to^*_{\mathcal{R},C} t$*.* □

Although conditional rewriting is decidable in our setting (strong determinism and quasi-decreasingness), we require a *conditional rewrite proof* to provide all the necessary information for checking a single conditional rewrite step (the employed rule, position,

and substitution; source and target terms; and recursively, a list of rewrite proofs for each condition of the applied rule). That way, we avoid having to formalize a rewriting engine for conditional rewriting in IsaFoR. With a check function for contextual rewrite sequences in place, we can easily give the check function *check-context-joinable* with the corresponding lemma:

Lemma 5.21. *Given a CTRS $\mathcal{R}$, three terms s, t, and u, a set of conditions C, and two lists of conditional rewrite proofs ps and qs, we have that*

`isOK` (*check-context-joinable u ps qs* $\mathcal{R}$ *s t C*)

implies that there exists some term u' such that $s \to^*_{\mathcal{R},C} u' \;{}^*_{\mathcal{R},C}\!\leftarrow t$. □

Here *check-context-joinable* is a concrete implementation of the homonymous function from the `al94-spec` locale. We further give the check function *check-unfeasible* and the accompanying soundness lemma:

Lemma 5.22. *Given a quasi-decreasing CTRS $\mathcal{R}$, two variable-disjoint variants of rules $\rho_1 \colon \ell_1 \to r_1 \Leftarrow c_1$ and $\rho_2 \colon \ell_2 \to r_2 \Leftarrow c_2$ in $\mathcal{R}$, an mgu μ of $\ell_1|_p$ and ℓ_2 for some position p, a set of conditions C such that $C = c_1\mu, c_2\mu$, three terms t, u, and v, and two lists of conditional rewrite proofs ps and qs, we have that*

`isOK` (*check-unfeasible t u v ps qs* ρ_1 ρ_2 $\mathcal{R}$ ℓ_1 μ *C*)

implies that there exist three terms t', u', and v' such that for all σ we have $\ell_1\mu\sigma \succ t'\sigma$, whenever σ satisfies C, $u' \;{}^*_{\mathcal{R},C}\!\leftarrow t' \to^*_{\mathcal{R},C} v'$, u' and v' are both strongly irreducible, and $u' \not\sim v'$. □

Again, *check-unfeasible* is a concrete implementation of the function of the same name from the `al94-spec` locale and it additionally performs various sanity checks.

At this point, interpreting the `al94-spec` locale using the three check functions *check-context-joinable*, *check-infeasible*, and *check-unfeasible* from above yields the concrete function *check-CCPs*, which is used in the final check *check-al94*.

Lemma 5.23. *Given a quasi-decreasing CTRS $\mathcal{R}$, a list of context-joinability certificates c, a list of infeasibility certificates i, and a list of unfeasibility certificates u. Then,* `isOK` (*check-al94 c i u* $\mathcal{R}$) *implies confluence of $\mathcal{R}$.* □

5.5 Chapter Notes

We have seen in this chapter how to extend the result that a terminating TRS is confluent if all its critical pairs are joinable to the conditional case. Unlike in the unconditional case this is still undecidable for CTRSs and instead of termination we employ quasi-decreasingness.

The original result (on which Theorem 5.6 is based) was published in 1994 by Avenhaus and Loría-Sáenz [3]. Their theorem employs quasi-reductivity instead of quasi-decreasingness, but unfortunately they do not use the definition of quasi-reductivity

common today (see Definition 6.1) but an older one due to Ganziger [18] that requires a reduction order instead of just a well-founded partial order that is closed under contexts. Closure under substitutions is used in the proof of [3, Theorem 4.2]. By a small change to the definition of unfeasibility we avoid this requirement for our extension to quasi-decreasingness. Unfeasibility and context-joinability have also already been introduced in the original paper. We added the third possibility that a critical pair may be infeasible.

Also Lemmas 5.8 and 5.10 are due to Avenhaus and Loría-Sáenz [3, Lemma 4.1(a,b) and Lemma 4.2] but the latter is stated as obvious without proof. We, however, deem the strengthened statement ($\star$) intricate enough to warrant a full proof (since without this strengthening, as far as we can tell, the outermost induction fails).

So in this chapter we described our formalization of a characterization of confluence of quasi-decreasing strongly deterministic CTRSs in Isabelle/HOL. It requires joinability of all conditional critical pairs, which is undecidable in general. Moreover, we formalized a more practical variant of the previous characterization for which each conditional critical pair must be either context-joinable, unfeasible, or infeasible. These properties, in turn, rely on strong irreducibility, which like strong determinism is undecidable in general. Thus, we further formalized decidable sufficient criteria.

In summary, this chapter constitutes the necessary work for the actual certification of confluence of quasi-decreasing SDCTRSs, which complements our other check functions for certifying confluence of CTRSs. We have extended our confluence tool ConCon and the certifier CeTA accordingly. The presentation in this chapter is mostly based on our publications [54, 60].

The CTRS in Example 5.1 is due to Bergstra and Klop [6, Example 3.6] (269) while the one in Example 5.2 is the oriented version of a join CTRS by Dershowitz et al. [13, Example B] (273 modulo a typo). The CTRSs of both, Example 5.3 as well as Example 5.4 are due to Avenhaus and Loría-Sáenz [3, Examples 4.1.a, 4.1.b] (262, 263). The CTRS in Example 5.15 is an adaptation of [35, Example 5.1] (292) from [54, Example 1], and finally the first CTRS in Example 5.13 is an inlined version (see Section 9.2) of [54, Example 3] (439).

The formalization of the methods described in this chapter can be found in the following IsaFoR theory files:

```
thys/Conditional_Rewriting/AL94.thy
thys/Conditional_Rewriting/AL94_Impl.thy
thys/Conditional_Rewriting/Quasi_Decreasingness.thy
```

To apply the method described in this chapter we first have to establish quasi-decreasingness of a CTRSs. This important property is the topic of the next chapter.

Chapter 6

Quasi-Decreasingness

In this chapter we will briefly look at quasi-decreasingness (see Definition 2.66), a property, that unlike effective termination, suffices to get effective computability of the rewrite relation of a CTRS (see Example 2.65). We are mainly interested in quasi-decreasingness because we will need it to apply the methods described in Chapter 5 but of course it is also interesting in its own right.

Actually, the results of this chapter employ the older notion of quasi-reductivity.

Definition 6.1 (Quasi-reductivity)**.** A DCTRS $\mathcal{R}$ over signature $\mathcal{F}$ is *quasi-reductive* if there is an extension $\mathcal{F}'$ of the signature (that is, $\mathcal{F} \subseteq \mathcal{F}'$) and a well-founded partial order $\succ$ on $\mathcal{T}(\mathcal{F}', \mathcal{V})$ that is closed under contexts such that for every substitution $\sigma\colon \mathcal{V} \to \mathcal{T}(\mathcal{F}', \mathcal{V})$ and every rule $\ell \to r \Leftarrow s_1 \approx t_1, \ldots, s_k \approx t_k$ in $\mathcal{R}$ we have that

- $s_j\sigma \succeq t_j\sigma$ for $1 \leqslant j < i$ implies $\ell\sigma \succ_{\mathsf{st}} s_i\sigma$ for all $1 \leqslant i \leqslant k$, and
- $s_j\sigma \succeq t_j\sigma$ for $1 \leqslant j \leqslant k$ implies $\ell\sigma \succ r\sigma$.

But it is well-known that quasi-reductivity implies quasi-decreasingness.

Lemma 6.2. *If a DCTRS $\mathcal{R}$ is quasi-reductive then $\mathcal{R}$ is quasi-decreasing.*

In contrast it is not known whether quasi-decreasingness differs from quasi-reductivity at all, that is, the question whether there exists a quasi-decreasing CTRS that is not quasi-reductive, is still open. Regardless, as put forward by Ohlebusch [49] quasi-decreasingness has two advantages over quasi-reductivity:

1. it does not depend on signature extensions and
2. $\ell\sigma \succ_{\mathsf{st}} s_i\sigma$ is only required if $s_j\sigma \to_{\mathcal{R}}^* t_j\sigma$ instead of $s_j\sigma \succeq t_j\sigma$.

Example 6.3. Point **1** is illustrated by the CTRS $\mathcal{R}$ consisting of the three rules

$$\mathsf{f}(\mathsf{b}) \to \mathsf{f}(\mathsf{a}) \qquad \mathsf{b} \to \mathsf{c} \qquad \mathsf{a} \to \mathsf{c} \Leftarrow \mathsf{b} \approx \mathsf{c}$$

over signature $\mathcal{F} = \{\mathsf{a}/0, \mathsf{b}/0, \mathsf{c}/0, \mathsf{f}/1\}$. We claim that $\mathcal{R}$ is quasi-decreasing. To show that we employ the TRS $\mathcal{U}$

$$\mathsf{f}(\mathsf{b}) \to \mathsf{f}(\mathsf{a}) \qquad \mathsf{b} \to \mathsf{c} \qquad \mathsf{a} \to \mathsf{U}(\mathsf{b})$$

This TRS is terminating (which can, for example, be shown by $\mathsf{TTT2}$). So the relation

$$\succ_1 \stackrel{\mathsf{def}}{=} (\to_{\mathcal{U}} \cup \rhd)^+ \cap (\mathcal{T}(\mathcal{F}, \mathcal{V}) \times \mathcal{T}(\mathcal{F}, \mathcal{V}))$$

on terms in $\mathcal{T}(\mathcal{F},\mathcal{V})$ is well-founded. Since $\mathsf{a} \to_{\mathcal{U}} \mathsf{U}(\mathsf{b}) \to_{\mathcal{U}} \mathsf{U}(\mathsf{c}) \rhd \mathsf{c}$ and $\mathsf{a} \to_{\mathcal{U}} \mathsf{U}(\mathsf{b}) \rhd \mathsf{b}$ we have $\to_{\mathcal{R}} \subseteq \succ_1$ and $\mathsf{a} \succ_1 \mathsf{b}$. Hence $\mathcal{R}$ is quasi-decreasing with respect to $\succ_1$. Now assume that $\mathcal{R}$ is quasi-reductive with respect to a relation $\succ_2$. From the first rule we get the restriction $\mathsf{f}(\mathsf{b}) \succ_2 \mathsf{f}(\mathsf{a})$ and from the last rule we get $\mathsf{a}\ (\succ_2 \cup \rhd)^+\ \mathsf{b}$. Since $\succ_2$ is closed under contexts we can use the property $\rhd \cdot \succ_2 \subseteq \succ_2 \cdot \rhd$ to shift all occurrences of $\rhd$ in $\mathsf{a}\ (\succ_2 \cup \rhd)^+\ \mathsf{b}$ to the end. If we only allow function symbols in $\mathcal{F}$, the latter sequence is either $\mathsf{a} \succ_2 \mathsf{b}$ or $\mathsf{a} \succ_2 \mathsf{f}^m(\mathsf{b}) \rhd \mathsf{b}$ for some $m \geqslant 1$. Together with closure under contexts and transitivity both of these contradict the well-foundedness of $\succ_2$. Hence $\mathcal{R}$ cannot be quasi-reductive (without extending the signature).

6.1 Characterizations

The following lemma employs transformation U (see Definition 2.49) to obtain quasi-reductivity.

Lemma 6.4. *A DCTRS $\mathcal{R}$ is quasi-reductive if $\mathsf{U}(\mathcal{R})$ is terminating.* □

Together with Lemma 6.2 we immediately get the following corollary, which we will prefer in the sequel.

Corollary 6.5. *A DCTRS $\mathcal{R}$ is quasi-decreasing if $\mathsf{U}(\mathcal{R})$ is terminating.* □

Example 6.6. Consider the DCTRS $\mathcal{R}$ consisting of the following two rules:

$$\mathsf{f}(x) \to x \Leftarrow x \approx \mathsf{a},\ \mathsf{b} \approx x \qquad\qquad \mathsf{a} \to \mathsf{b}$$

The unraveled TRS $\mathsf{U}(\mathcal{R})$ comprises the four rules

$$\mathsf{f}(x) \to \mathsf{U}_1(x,x) \qquad \mathsf{U}_1(\mathsf{a},x) \to \mathsf{U}_2(\mathsf{b},x) \qquad \mathsf{U}_2(x,x) \to x \qquad \mathsf{a} \to \mathsf{b}$$

It is shown to be terminating by the following polynomial interpretation $\mathcal{I}$:

$$\begin{aligned} \mathsf{a}_{\mathcal{I}} &= 1 & \mathsf{U}_{1\mathcal{I}}(x,y) &= x+y+1 \\ \mathsf{b}_{\mathcal{I}} &= 0 & \mathsf{U}_{2\mathcal{I}}(x,y) &= x+y+1 \\ \mathsf{f}_{\mathcal{I}}(x) &= 2x+2 \end{aligned}$$

because we have

$$\begin{aligned} \mathsf{f}_{\mathcal{I}}(x) &= 2x+2 > 2x+1 = \mathsf{U}_{1\mathcal{I}}(x,x) \\ \mathsf{U}_{1\mathcal{I}}(\mathsf{a}_{\mathcal{I}},x) &= x+2 > x+1 = \mathsf{U}_{2\mathcal{I}}(\mathsf{b}_{\mathcal{I}},x) \\ \mathsf{U}_{2\mathcal{I}}(x,x) &= 2x+1 > x \\ \mathsf{a}_{\mathcal{I}} &= 1 > 0 = \mathsf{b}_{\mathcal{I}} \end{aligned}$$

for all natural numbers x. Hence $\mathcal{R}$ is quasi-decreasing by Corollary 6.5.

Also transformation V (see Definition 2.51) can be employed to obtain get quasi-reductivity as shown in the following lemma.

Lemma 6.7. *A DCTRS $\mathcal{R}$ is quasi-reductive if $\mathsf{V}(\mathcal{R})$ is simply terminating.* □

Again, we will rather use the corresponding corollary.

Corollary 6.8. *A DCTRS $\mathcal{R}$ is quasi-decreasing if $\mathsf{V}(\mathcal{R})$ is simply terminating.* □

Example 6.9. Consider the DCTRS $\mathcal{R}$ consisting of the two rules

$$\mathsf{f}(x) \to \mathsf{a} \Leftarrow \mathsf{a} \approx x \qquad\qquad \mathsf{f}(x) \to \mathsf{b} \Leftarrow \mathsf{b} \approx x$$

The TRS $\mathsf{V}(\mathcal{R})$ has only two rules

$$\mathsf{f}(x) \to \mathsf{a} \qquad\qquad \mathsf{f}(x) \to \mathsf{b}$$

In order to ensure the subterm property, that we need to show simple termination, we extend this TRS (over signature $\mathcal{F}$) by the TRS $\mathcal{E}\mathsf{mb}(\mathcal{F})$ that in this case consists of the single rule $\mathsf{f}(x) \to x$. Now we have to show termination of the three-rule TRS

$$\mathsf{f}(x) \to \mathsf{a} \qquad\qquad \mathsf{f}(x) \to \mathsf{b} \qquad\qquad \mathsf{f}(x) \to x$$

This can be done by the following polynomial interpretation $\mathcal{I}$

$$\mathsf{a}_{\mathcal{I}} = 0 \qquad\qquad \mathsf{b}_{\mathcal{I}} = 0 \qquad\qquad \mathsf{f}_{\mathcal{I}}(x) = x + 1$$

because we have

$$\mathsf{f}_{\mathcal{I}}(x) = x + 1 > 0 = \mathsf{a}_{\mathcal{I}} \qquad \mathsf{f}_{\mathcal{I}}(x) = x + 1 > 0 = \mathsf{b}_{\mathcal{I}} \qquad \mathsf{f}_{\mathcal{I}}(x) = x + 1 > x$$

Since $\mathsf{V}(\mathcal{R}) \cup \mathcal{E}\mathsf{mb}(\mathcal{F})$ is terminating, $\mathsf{V}(\mathcal{R})$ is simply terminating and hence $\mathcal{R}$ is quasi-decreasing by Corollary 6.8.

What both of the above characterizations of quasi-decreasingness have in common is that they cannot be used to show that a CTRS is *not* quasi-decreasing. To do that we have to restrict the possible rewrite sequences in the unraveled system in order to more closely capture the sequences that are possible in the original CTRS. Schernhammer and Gramlich use context-sensitivity to extend the usual unraveling U by a replacement map in order to restrict reductions in U-symbols to the first argument position.

Definition 6.10 (Context-sensitive unraveling U_{CS})**.** The *context-sensitive unraveling* $\mathsf{U}_{\mathsf{CS}}(\mathcal{R})$ is the unraveling $\mathsf{U}(\mathcal{R})$ together with the replacement map μ such that $\mu(f) = \{1, \ldots, k\}$ if $f \in \mathcal{F}$ with arity k and $\mu(f) = \{1\}$ otherwise.

To arrive at the characterization below we additionally have to restrict our notion of termination.

Definition 6.11 (μ-termination on original terms)**.** Given a DCTRS $\mathcal{R}$ over signature $\mathcal{F}$ we say that the CSRS $\mathsf{U}_{\mathsf{CS}}(\mathcal{R})$ is *μ-terminating on original terms*, if there is no infinite $\mathsf{U}_{\mathsf{CS}}(\mathcal{R})$-reduction starting from a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$.

Theorem 6.12. *Quasi-decreasingness of a DCTRS $\mathcal{R}$ is equivalent to μ-termination of the CSRS $\mathsf{U}_{\mathsf{CS}}(\mathcal{R})$ on original terms.* □

While the direction that quasi-decreasingness of $\mathcal{R}$ implies μ-termination of $\mathsf{U}_{\mathsf{CS}}(\mathcal{R})$ on original terms is shown directly by Schernhammer and Gramlich; the converse employs the additional notion of context-sensitive quasi-reductivity of $\mathcal{R}$.

Definition 6.13 (Context-sensitive quasi-reductivity)**.** A CSRS $\mathcal{R}$ over signature $\mathcal{F}$ is *context-sensitively quasi-reductive* if there is an extended signature $\mathcal{F}' \supseteq \mathcal{F}$, a replacement map μ (with $\mu(f) = \{1, \ldots, n\}$ for $f/n \in \mathcal{F}$), and a μ-monotonic, well-founded partial order $\succ_\mu$ on $\mathcal{T}(\mathcal{F}', \mathcal{V})$ such that for every rule $\ell \to r \Leftarrow s_1 \approx t_1, \ldots, s_k \approx t_k$ and every substitution $\sigma : \mathcal{V} \to \mathcal{T}(\mathcal{F}, \mathcal{V})$:

- $s_j\sigma \succeq_\mu t_j\sigma$ for $1 \leqslant j < i$ implies $\ell\sigma \; (\succ_\mu \cup \rhd_\mu)^+ \; s_i\sigma$ for all $1 \leqslant i \leqslant k$, and
- $s_j\sigma \succeq_\mu t_j\sigma$ for $1 \leqslant j \leqslant k$ implies $\ell\sigma \succ_\mu r\sigma$.

In the following, we give a direct proof of the fact that μ-termination of $\mathsf{U}_{\mathsf{CS}}(\mathcal{R})$ on original terms implies quasi-decreasingness of $\mathcal{R}$ avoiding the complicated notion above. Our proof employs reduction-preservation of U_{CS} (that is, every $\mathcal{R}$-step can be simulated by a $\mathsf{U}_{\mathsf{CS}}(\mathcal{R})$-reduction), that can be shown by induction on the level of a conditional rewrite step.

Theorem 6.14 (Reduction-preservation of U_{CS})**.** *For every DCTRS $\mathcal{R}$ the inclusion $\to_{\mathcal{R}} \subseteq \to^+_{\mathsf{U}_{\mathsf{CS}}(\mathcal{R})}$ holds.* □

Beyond that, we also employ the following auxiliary result.

Lemma 6.15. *For any context-sensitive rewrite relation $\to_\mu$ induced by the replacement map μ, $\rhd_\mu$ commutes over $\to_\mu$, that is, $\rhd_\mu \cdot \to_\mu \subseteq \to_\mu \cdot \rhd_\mu$.*

Proof. Assume $s \rhd_\mu t \to_\mu u$ for some terms s, t, and u. Then $s = C[t]_p \rhd_\mu t \to_\mu u$ for some non-empty context C and active position p. Thus, because $\to_\mu$ is closed under contexts at active positions, we conclude by $C[t]_p \to_\mu C[u]_p \rhd_\mu u$. □

With this we are finally able to give our direct proof for the following theorem:

Theorem 6.16. *If the CSRS $\mathsf{U}_{\mathsf{CS}}(\mathcal{R})$ is μ-terminating on original terms then the DCTRS $\mathcal{R}$ is quasi-decreasing.*

Proof. Assume that $\mathsf{U}_{\mathsf{CS}}(\mathcal{R})$ is μ-terminating on original terms. We define an order $\succ$ on terms $\mathcal{T}(\mathcal{F}, \mathcal{V})$

$$\succ \stackrel{\mathsf{def}}{=} (\to_{\mathsf{U}_{\mathsf{CS}}(\mathcal{R})} \cup \rhd_\mu)^+ \cap (\mathcal{T}(\mathcal{F}, \mathcal{V}) \times \mathcal{T}(\mathcal{F}, \mathcal{V})) \qquad (\star)$$

and show that it satisfies the four properties from the definition of quasi-decreasingness (see Definition 2.66).

1. We start by showing that $\succ$ is well-founded on $\mathcal{T}(\mathcal{F},\mathcal{V})$. Assume, to the contrary, that $\succ$ is not well-founded. Then we have an infinite sequence

$$t_1 \succ t_2 \succ t_3 \succ \cdots \tag{$\dagger$}$$

   where all $t_i \in \mathcal{T}(\mathcal{F},\mathcal{V})$. By definition $\rhd_\mu$ is well-founded. Moreover, since $\mathsf{U}_{\mathsf{CS}}(\mathcal{R})$ is μ-terminating on original terms, $\to_{\mathsf{U}_{\mathsf{CS}}(\mathcal{R})}$ is well-founded on $\mathcal{T}(\mathcal{F},\mathcal{V})$. Further note that every $\to_{\mathsf{U}_{\mathsf{CS}}(\mathcal{R})}$-terminating element (hence every term in $\mathcal{T}(\mathcal{F},\mathcal{V})$) is $\to_{\mathsf{U}_{\mathsf{CS}}(\mathcal{R})/\rhd_\mu}$-terminating, since by a repeated application of Lemma 6.15 every infinite reduction $t_1 \to_{\mathsf{U}_{\mathsf{CS}}(\mathcal{R})/\rhd_\mu} t_2 \to_{\mathsf{U}_{\mathsf{CS}}(\mathcal{R})/\rhd_\mu} \cdots$ starting from a term $t_1 \in \mathcal{T}(\mathcal{F},\mathcal{V})$ can be transformed into an infinite $\to_{\mathsf{U}_{\mathsf{CS}}(\mathcal{R})}$-reduction, contradicting well-foundedness of $\to_{\mathsf{U}_{\mathsf{CS}}(\mathcal{R})}$ on $\mathcal{T}(\mathcal{F},\mathcal{V})$. We conclude by analyzing the following two cases:
   - Either ($\dagger$) contains $\to_{\mathsf{U}_{\mathsf{CS}}(\mathcal{R})}$ only finitely often, contradicting well-foundedness of $\rhd_\mu$,
   - or there are infinitely many $\to_{\mathsf{U}_{\mathsf{CS}}(\mathcal{R})}$-steps in ($\dagger$). But then we can construct a sequence $s_1 \to_{\mathsf{U}_{\mathsf{CS}}(\mathcal{R})/\rhd_\mu} s_2 \to_{\mathsf{U}_{\mathsf{CS}}(\mathcal{R})/\rhd_\mu} s_3 \to_{\mathsf{U}_{\mathsf{CS}}(\mathcal{R})/\rhd_\mu} \cdots$ with $s_1 = t_1$, contradicting the fact that all elements of $\mathcal{T}(\mathcal{F},\mathcal{V})$ are $\to_{\mathsf{U}_{\mathsf{CS}}(\mathcal{R})/\rhd_\mu}$-terminating.
2. Next we show $\succ = (\succ \cup \rhd)^+$. The direction $\succ \subseteq (\succ \cup \rhd)^+$ is obvious. For the other direction, $(\succ \cup \rhd)^+ \subseteq \succ$, assume we have $s\ (\succ \cup \rhd)^{n+1}\ t$. Then we proceed by induction on n. In the base case $s\ (\succ \cup \rhd)\ t$. If $s \succ t$ we are done. Otherwise, $s \rhd t$ and thus also $s \rhd_\mu t$ since $s,t \in \mathcal{T}(\mathcal{F},\mathcal{V})$ and therefore $s \succ t$. In the step case $n = m+1$ for some m, and $s\ (\succ \cup \rhd)\ u\ (\succ \cup \rhd)^m\ t$. Then we obtain $s \succ u$ by a similar case-analysis as in the base case. Moreover $u \succ t$ by induction hypothesis, and thus $s \succ t$.
3. Now we show that $\to_{\mathcal{R}} \subseteq \succ$. Assume $s \to_{\mathcal{R}} t$. Together with reduction-preservation of $\mathsf{U}_{\mathsf{CS}}(\mathcal{R})$, Theorem 6.14, we get $s \to^+_{\mathsf{U}_{\mathsf{CS}}(\mathcal{R})} t$ which in turn implies $s \succ t$.
4. Finally, we show that if for all $\ell \to r \Leftarrow s_1 \approx t_1, \ldots, s_k \approx t_k$ in $\mathcal{R}$, substitutions $\sigma\colon \mathcal{V} \to \mathcal{T}(\mathcal{F},\mathcal{V})$, and $1 \leqslant i \leqslant k$, if $s_j\sigma \to^*_{\mathcal{R}} t_j\sigma$ for all $1 \leqslant j < i$ then $\ell\sigma \succ s_i\sigma$. We have the sequence

$$\ell\sigma \to^+_{\mathsf{U}_{\mathsf{CS}}(\mathcal{R})} U^\rho_i(s_i, \mathsf{var}(\ell), \mathsf{ev}(t_1,\ldots,t_i))\sigma \rhd_\mu s_i\sigma$$

   using the definition of $\mathsf{U}_{\mathsf{CS}}(\mathcal{R})$ together with reduction-preservation (Theorem 6.14). But then also $\ell\sigma \succ s_i\sigma$ as wanted because $\ell\sigma, s_i\sigma \in \mathcal{T}(\mathcal{F},\mathcal{V})$.

Hence $\mathcal{R}$ is quasi-decreasing with the order $\succ$. □

6.2 Certification and Implementation

From the above results – Corollary 6.5, Corollary 6.8, and Theorem 6.12 – only the first is formalized in IsaFoR. We still utilize Corollary 6.8 in ConCon but only if we do not restrict to certifiable methods (see Chapter 10). To avoid having to employ complicated setups for external termination tools to ensure simple termination of $\mathsf{V}(\mathcal{R})$ in our implementation of Corollary 6.8 we rather extend the transformed TRS by the embedding rules $\mathcal{E}\mathsf{mb}(\mathcal{F})$ depending on $\mathcal{R}$'s signature $\mathcal{F}$ (as shown in Example 6.9) to guarantee the subterm property.

6.3 Chapter Notes

In this chapter we have seen two lemmas using different transformations that allow to reduce quasi-decreasingness of a CTRS to (simple) termination of the transformed TRS. Moreover, we provide a direct proof for one direction of a full characterization of quasi-decreasingness (see Theorem 6.16), that is, μ-termination of a CSRS $\mathsf{U}_{\mathsf{CS}}(\mathcal{R})$ on original terms implies quasi-decreasingness of the DCTRS $\mathcal{R}$ without the need of a detour by using the notion of context-sensitive quasi-reductivity. We believe that our proof could easily be adapted to any other context-sensitive transformation as long as it is reduction-preserving. Knowing that a DCTRS is quasi-decreasing is, among other things, useful to show confluence with the criterion of Chapter 5.

Lemma 6.4 is from [49, p. 214] while Lemma 6.7 is a combination of [49, Lemma 7.2.6] and [49, Proposition 7.2.68]. It is unknown whether the condition of Lemma 6.4 is implied by the condition of Lemma 6.7 (see [49, p. 229]). The context-sensitive version of unraveling U was originally defined by Schernhammer and Gramlich [51, Definition 4] who also coined μ-termination on original terms [51, Definition 7] and proved Theorem 6.14 [51, Theorem 1] and Theorem 6.12. That a DCTRS is quasi-decreasing if its unraveling is terminating has been formalized by Winkler and Thiemann [66] in IsaFoR. The other two results, Corollary 6.8 and Theorem 6.12, are not formalized yet. This is basically because we just did not find the time to investigate how difficult it would be to do that. Since IsaFoR does not even provide the first result about context-sensitive rewriting yet, we expect that formalizing Theorem 6.12 would be quite involved.

In 2005 Lucas et al. introduced the notion of *operational termination* which is basically defined as the absence of infinite derivation trees with respect to the inference rules of conditional rewriting logic. For DCTRSs quasi-decreasingness and operational termination are equivalent [37]. Recently the dependency pair framework has been adapted for operational termination [38].

The CTRS in Example 6.6 is from [48, Example 3.5] (`317`), the one in Example 6.9 is from [59, Example 3.3] (`492`), and the one in Example 6.3 is from [54].

The formalization of the methods described in this chapter can be found in the following IsaFoR theory files:

```
thys/Conditional_Rewriting/Quasi_Decreasingness.thy
thys/Conditional_Rewriting/Unraveling.thy
thys/Conditional_Rewriting/Unraveling_Impl.thy
```

The next chapter discusses several methods to show infeasibility of conditional critical pairs. Specifically the techniques described in Chapters 4 and 5 benefit from these infeasibility results.

Chapter 7

Infeasibility of Conditional Critical Pairs

The confluence methods detailed in Chapters 4 and 5 among other things also analyze the conditional critical pairs of a CTRS. Being able to ignore certain critical pairs simplifies this analysis. Now, if the conditions of a CCP are not satisfiable then the resulting equation can never be utilized and hence is harmless for the confluence of the CTRS under consideration. Such CCPs (with unsatisfiable conditions) are called *infeasible* (see Definition 2.75) and we can safely ignore them for the purpose of confluence.

Example 7.1. For instance, the single CCP (modulo symmetry)

$$\mathsf{h}(\mathsf{a}, x) \approx \mathsf{h}(\mathsf{f}(x, y), z) \Leftarrow \mathsf{g}(\mathsf{a}) \approx \mathsf{h}(y, z)$$

of the oriented 3-CTRS $\mathcal{R}$ consisting of the two rules

$$\mathsf{g}(\mathsf{f}(x, \mathsf{a})) \to \mathsf{h}(\mathsf{a}, y) \qquad \mathsf{g}(\mathsf{f}(x, y)) \to \mathsf{h}(\mathsf{f}(x, z), v) \Leftarrow \mathsf{g}(y) \approx \mathsf{h}(z, v)$$

is infeasible since $\mathsf{g}(\mathsf{a})$ is a normal form. Hence $\mathcal{R}$ is orthogonal (modulo infeasibility) and thus confluent.

In this chapter we present an overview of infeasibility methods for oriented 3-CTRSs, one of the most popular types of conditional rewriting. In such systems extra variables in conditions and right-hand sides of rewrite rules are allowed to a certain extend (see Definition 2.38). Moreover, for oriented CTRSs satisfiability of the conditions amounts to reachability (see page 18). As a consequence of the latter, establishing infeasibility is similar to the problem of eliminating arrows in dependency graph approximations, a problem which has been investigated extensively in the literature. The difference is that we deal with CTRSs and the terms we test may share variables.

For brevity, we speak about non-reachability, non-meetability, and non-joinability of two terms s and t, when we actually mean that the respective property holds for arbitrary substitution instances $s\sigma$ and $t\tau$.

In the sequel we summarize the methods that we have analyzed and adapted for infeasibility.

7.1 Unification

A widely used method to check for non-reachability is to try to unify the tcap of the source term with the target term; which is the de facto standard for approximating dependency

graphs for termination proofs. Remember, the tcap-function approximates the topmost part of a term, its "cap", that does not change under rewriting (see Definition 2.33). It is well known that $\mathsf{tcap}(s) \not\sim t$ implies non-reachability of t from s. Typical "pen and paper" definitions (like the one in the preliminaries) rely on replacing subterms by "fresh variables". Instead of inventing fresh variables out of thin air, the IsaFoR-version of tcap replaces every variable occurrence by the symbol $\Box$. The resulting terms behave like ground multihole contexts – we call them *ground contexts* – and they are intended to represent the set of all terms resulting from replacing all "holes" by arbitrary terms. This is made formal by the *substitution instance class* of a ground context.

Definition 7.2 (Substitution instance class)**.** The *substitution instance class* $[\![t]\!]$ of a ground context t is defined recursively by.

$$[\![t]\!] := \begin{cases} \mathcal{T}(\mathcal{F}, \mathcal{V}) & \text{if } t = \Box \\ \{f(s_1, \ldots, s_n) \mid s_i \in [\![t_i]\!]\} & \text{if } t = f(t_1, \ldots, t_n) \end{cases}$$

Note that for variable-disjoint terms s and t, unifiability coincides with $s\sigma = t\tau$ for some, not necessarily identical, substitutions σ and τ. Thus asking whether a term t unifies with a variable-disjoint term represented by the ground context s is equivalent to checking whether $t\sigma \in [\![s]\!]$ for some substitution σ. The latter is called *ground context matching*. Thus we can define an efficient (see Section 7.8) executable version of tcap by:

Definition 7.3 (Efficient tcap)**.**

$$\mathsf{tcap}_{\mathcal{R}}(t) = \begin{cases} \Box & \text{if } t \text{ is a variable} \\ \Box & \text{if } t = f(t_1, \ldots, t_n) \text{ and } \ell\sigma \in [\![u]\!] \text{ for some } \sigma \text{ and } \ell \to r \in \mathcal{R} \\ u & \text{otherwise} \end{cases}$$

where $u = f(\mathsf{tcap}_{\mathcal{R}}(t_1), \ldots, \mathsf{tcap}_{\mathcal{R}}(t_n))$ and $\mathcal{R}$ is a TRS. We omit $\mathcal{R}$ if it is clear from context.

This version of tcap is sound, that is, whenever we can reach a term t from an instance of a term s then t is in the substitution instance class of $\mathsf{tcap}(s)$.

Lemma 7.4. *If* $s\sigma \to_{\mathcal{R}}^* t$ *then* $t \in [\![\mathsf{tcap}(s)]\!]$. □

Then checking non-reachability of t from s amounts to deciding whether there does not exist a substitution τ such that $t\tau \in [\![\mathsf{tcap}(s)]\!]$, for which we use the more succinct notation $\mathsf{tcap}(s) \not\sim t$ almost everywhere else in this thesis.

While the above definition of tcap and the corresponding soundness lemma were already present in IsaFoR, the following easy extension also allows us to test for non-joinability.

Lemma 7.5. *If* $s\sigma \to_{\mathcal{R}}^* \cdot {}_{\mathcal{R}}^*\!\leftarrow t\tau$ *then* $[\![\mathsf{tcap}(s)]\!] \cap [\![\mathsf{tcap}(t)]\!] \neq \varnothing$.

Proof. We have $s\sigma \to_{\mathcal{R}}^* u$ and $t\tau \to_{\mathcal{R}}^* u$ for some u. By Lemma 7.4 we have $u \in \mathsf{tcap}(s)$ and $u \in \mathsf{tcap}(t)$. □

Fortunately the same techniques that are used to obtain an algorithm for ground context matching can be reused for *ground context unifiability*, that is, checking $[\![\mathsf{tcap}(s)]\!] \cap [\![\mathsf{tcap}(t)]\!] \neq \varnothing$ (elsewhere in this thesis we use the notation $\mathsf{tcap}(s) \not\sim \mathsf{tcap}(t)$).

Now for checking infeasibility of a CCP we use the underlying TRS and if there is more than one condition we collect the left- and right-hand sides separately in a fresh function symbol. This gives the following corollary (where cs is a fresh function symbol of arity k).

Corollary 7.6 (Infeasibility via tcap)**.** *Let $\mathcal{R}$ be an oriented 3-CTRS. A CCP*

$$u \approx v \Leftarrow s_1 \approx t_1, \ldots, s_k \approx t_k$$

is infeasible if $\mathsf{tcap}_{\mathcal{R}_\mathsf{u}}(\mathsf{cs}(s_1, \ldots, s_k)) \not\sim \mathsf{cs}(t_1, \ldots, t_k)$.

Example 7.7. Consider the CTRS $\mathcal{R}$ from Example 7.1. The CCP

$$\mathsf{h}(\mathsf{a}, x) \approx \mathsf{h}(\mathsf{f}(x, y), z) \Leftarrow \mathsf{g}(\mathsf{a}) \approx \mathsf{h}(y, z)$$

is infeasible by Corollary 7.6 because $\mathsf{tcap}_{\mathcal{R}_\mathsf{u}}(\mathsf{g}(\mathsf{a})) = \mathsf{g}(\mathsf{a}) \not\sim \mathsf{h}(x, v) = \mathsf{tcap}_{\mathcal{R}_\mathsf{u}}(\mathsf{h}(y, z))$.

7.2 Symbol Transition Graph

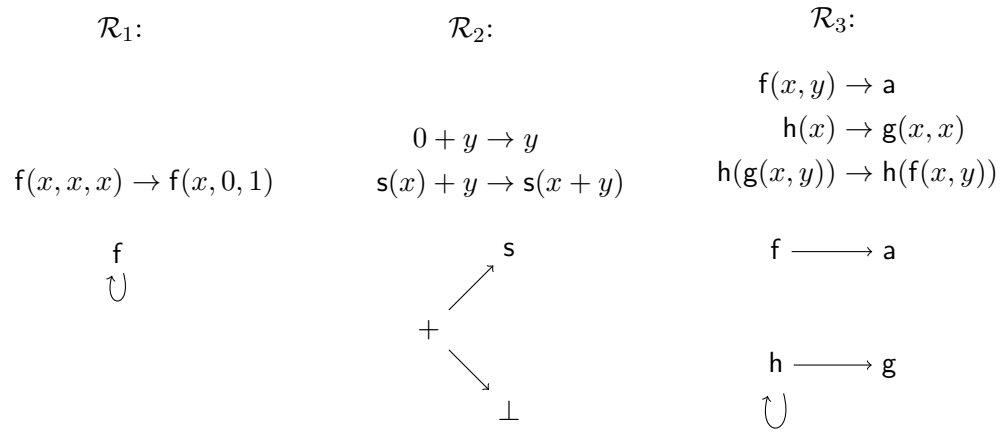
One shortcoming of the tcap method is that although later during unification we have to consider the target term anyway it first only considers the starting term. Maybe we could gain power if we use knowledge about the structure of the target from the start? This consideration is the basis for the so called *symbol transition graph*. The idea is simple enough, we look at the root symbols (see Definition 2.18) of the left- and right-hand sides of the rewrite rules of a TRS (for CTRSs we just approximate by using the underlying TRS $\mathcal{R}_\mathsf{u}$) and we collect the dependencies in a graph.

Definition 7.8 (Symbol transition graph)**.** Given a TRS $\mathcal{R}$ the edges of its *symbol transition graph* are given by the relation

$$\sqsupset_{\mathsf{stg}} := \{(\mathsf{root}(\ell), \mathsf{root}(r)) \mid \ell \to r \in \mathcal{R}\}$$

To clarify this concept let us first look at some examples.

Example 7.9. Below we depict the symbol transition graphs of the following three TRSs $\mathcal{R}_1$, $\mathcal{R}_2$, and $\mathcal{R}_3$, respectively.



Example 7.10. For the CTRS of Example 7.1 the symbol transition graph consists of a single arrow:

$$\mathsf{g} \longrightarrow \mathsf{h}$$

Now we can use the following lemma for reachability analysis of TRSs.

Lemma 7.11. *For two terms $s = f(\ldots)$ and $t = g(\ldots)$ if $s \to^* t$ then either $f = g$, $f \sqsupset^+_{\mathsf{stg}} g$, or $f \sqsupset^+_{\mathsf{stg}} \bot$.*

Proof. We prove this by induction on the length of the rewrite sequence $s \to^k t$. In the base case we have $s = t$ and hence $f = g$. Now in the step case we look at the sequence $s \to u \to^k t$. If u is a variable then $s \to u$ is a root step with a collapsing rule and hence $f \sqsupset_{\mathsf{stg}} \bot$ and thus also $f \sqsupset^+_{\mathsf{stg}} \bot$. Otherwise there is some function symbol h such that $u = h(\ldots)$. If we have a non-root step then $f = h$ and we are done. Otherwise we have $f \sqsupset_{\mathsf{stg}} h$. From the induction hypothesis we know that one of $h = g$, $h \sqsupset^+_{\mathsf{stg}} g$, or $h \sqsupset^+_{\mathsf{stg}} \bot$ holds. So either $f \sqsupset_{\mathsf{stg}} h = g$, $f \sqsupset_{\mathsf{stg}} h \sqsupset^+_{\mathsf{stg}} g$, or $f \sqsupset_{\mathsf{stg}} h \sqsupset^+_{\mathsf{stg}} \bot$, which finishes the proof. □

From the previous lemma we immediately get the following non-reachability result:

Corollary 7.12. *If $f \neq g$ and neither $f \sqsupset^+_{\mathsf{stg}} g$ nor $f \sqsupset^+_{\mathsf{stg}} \bot$ then $f(\ldots) \not\to^* g(\ldots)$.*

Example 7.13. Consider the TRS $\mathcal{R}_3$ from Example 7.9. Do we have a rewrite sequence $\mathsf{f}(x,y)\sigma \to^* \mathsf{g}(x,y)\tau$ for some substitutions σ and τ? Since $\mathsf{tcap}(\mathsf{f}(x,y)) = z \sim \mathsf{g}(x,y)$ we cannot conclude non-reachability using tcap. Then again $\mathsf{f} \neq \mathsf{g}$, $\mathsf{f} \not\sqsupset^+_{\mathsf{stg}} \mathsf{g}$, and $\mathsf{f} \not\sqsupset^+_{\mathsf{stg}} \bot$. So $\mathsf{g}(x,y)$ is not reachable from $\mathsf{f}(x,y)$ with respect to $\mathcal{R}_3$ by Corollary 7.12.

Example 7.14. For the CTRS of Example 7.1 to get infeasibility of its CCP we have to show that either $x\sigma \not\to^* \mathsf{a}\tau$, $x\sigma \not\to^* \mathsf{c}\tau$, or even $\mathsf{cs}(x,x)\sigma \not\to^* \mathsf{cs}(\mathsf{a},\mathsf{c})\tau$ for a fresh compound symbol cs and any two substitutions σ and τ. Unfortunately we cannot employ Corollary 7.12 here. If we look at the two conditions separately the left-hand sides are variables and looking at the combined conditions we have the same function symbol cs on the left- and right-hand sides.

In the next section we will see a way to combine the best of both tcap and the symbol transition graph.

7.3 Decomposing Reachability Problems

Both, the tcap method and the symbol transition graph do have their pros and cons. For example, in contrast to the tcap method, the symbol transition graph takes information about the target term into account, then again, it does not recursively look at the whole term, like tcap, but only at the root symbols.

In this section we want to introduce a modular framework for reachability, that allows to decompose a given problem into several smaller ones. The binary relation $\sqsupset$ that we employ to show non-reachability between two terms has to have certain properties to be usable for decomposition. These properties are summarized in the following definition.

Definition 7.15 (Decomposition)**.** A binary relation on terms $\sqsupset$ *admits decomposition* if both

1. $s\sigma \sqsupset t\tau$ implies $s \sqsupset t$ for all substitutions σ and τ and
2. $s \not\sqsupset t$ and $s \to^* t$ implies that there is no single root step in $s \to^* t$.

Using the binary relation $\sqsupset$ we can define a *decomposition function* that deconstructs a single reachability problem into several (smaller) ones.

Definition 7.16 (Abstract decomposition function)**.** The *abstract decomposition function* $\mathsf{D}_{s,t}$ takes a reachability problem (s,t) (meaning: "Is t reachable from s?") as input and recursively computes a set of (possibly easier) reachability problems with respect to the relation $\sqsupset$.

$$\mathsf{D}_{s,t} := \begin{cases} \mathsf{D}_{s_1,t_1} \cup \ldots \cup \mathsf{D}_{s_n,t_n} & \text{if } s = f(s_1,\ldots,s_n), t = f(t_1,\ldots,t_n), \text{ and } s \not\sqsupset t \\ \{(s,t)\} & \text{otherwise} \end{cases}$$

With this function in place we can now formally define when decomposition is admissible for a reachability problem.

Lemma 7.17. *If $\sqsupset$ admits decomposition and $s\sigma \to^* t\tau$ then also $u\sigma \to^* v\tau$ for all $(u,v) \in \mathsf{D}_{s,t}$.* □

Of course in our setting we want to use decomposition of a reachability problem to show non-reachability. To do that we first have to specify an abstract non-reachability test that we will later instantiate with concrete checks.

Definition 7.18 (Abstract non-reachability check)**.** If two functional terms $s = f(\ldots)$ and $t = g(\ldots)$ have different root symbols and s and t are not in the relation $\sqsupset$ then t is not reachable from s. Otherwise it is. This is tested by the *abstract non-reachability check* $\mathsf{nonreach}(s,t)$ defined as follows:

$$\mathsf{nonreach}(s,t) := \begin{cases} f \neq g \wedge s \not\sqsupset t & \text{if } s = f(\ldots) \text{ and } t = g(\ldots) \\ \mathsf{false} & \text{otherwise} \end{cases}$$

Finally, we are ready to state a non-reachability lemma.

Lemma 7.19. *If $\sqsupset$ admits decomposition and* $\mathsf{nonreach}(s,t)$ *holds then* $s\sigma \not\to^* t\tau$.

Proof. Assume to the contrary that $s\sigma \to^* t\tau$. Because $\sqsupset$ admits decomposition we have

1. $s\sigma \sqsupset t\tau \Longrightarrow s \sqsupset t$
2. $s \not\sqsupset t \Longrightarrow s \to^* t \Longrightarrow s \xrightarrow{>\epsilon}{}^* t$

From **1** and $\mathsf{nonreach}(s,t)$ we have that $s\sigma \not\sqsupset t\tau$. From this, our starting assumption, and **2** we have $s\sigma \xrightarrow{>\epsilon}{}^* t\tau$. But this implies $\mathsf{root}(s) = \mathsf{root}(t)$ which contradicts $\mathsf{nonreach}(s,t)$. □

Below we give two possible instances of $\sqsupset$ one using the tcap-function of Section 7.1 and the other using the symbol transition graph $\sqsupset^+_{\mathsf{stg}}$ of Section 7.2.

Definition 7.20 ($\sqsupset_{\mathsf{tcap}}$, $\sqsupset_{\mathsf{rs}}$)**.** We define

- $s \sqsupset_{\mathsf{tcap}} t$ iff there exists a rule $\ell \to r$ in $\mathcal{R}$ such that $\mathsf{tcap}(s)$ and ℓ are unifiable and
- $s \sqsupset_{\mathsf{rs}} t$ iff either s is a variable, t is a variable, $\mathsf{root}(s) \sqsupset^+_{\mathsf{stg}} \mathsf{root}(t)$, or $\mathsf{root}(s) \sqsupset^+_{\mathsf{stg}} \bot$.

Example 7.21. Consider the TRS $\mathcal{R}_4$ consisting of the four rules

$$\mathsf{f}(x) \to \mathsf{a} \qquad \mathsf{p}(\mathsf{s}(x)) \to \mathsf{b} \qquad \mathsf{h}(x) \to \mathsf{g}(x) \qquad \mathsf{h}(\mathsf{f}(\mathsf{b})) \to \mathsf{h}(\mathsf{f}(\mathsf{p}(\mathsf{a})))$$

and the two terms $s = \mathsf{f}(\mathsf{p}(\mathsf{a}))$ and $t = \mathsf{f}(\mathsf{b})$. We have $s \sqsupset_{\mathsf{tcap}} t$ since $\mathsf{tcap}(s) = y$ unifies with the left-hand side of every rule in $\mathcal{R}_4$ but $s \not\sqsupset_{\mathsf{rs}} t$ because s and t are not variables and neither $\mathsf{f} \sqsupset^+_{\mathsf{stg}} \mathsf{f}$ nor $\mathsf{f} \sqsupset^+_{\mathsf{stg}} \bot$. Moreover, for the two terms $s' = \mathsf{p}(\mathsf{a})$ and $t' = \mathsf{b}$ we have $s' \not\sqsupset_{\mathsf{tcap}} t'$ since $\mathsf{tcap}(s') = \mathsf{p}(\mathsf{a})$ does not unify with the left-hand side of any rule in $\mathcal{R}_4$ but $s' \sqsupset_{\mathsf{rs}} t'$ because the second rule of $\mathcal{R}_4$ yields $\mathsf{p} \sqsupset_{\mathsf{stg}} \mathsf{b}$.

Both of these relations admit decomposition of reachability problems.

Lemma 7.22. *$\sqsupset_{tcap}$ and $\sqsupset_{\mathsf{rs}}$ admit decomposition.*

Proof. We first consider $\sqsupset_{\mathsf{tcap}}$. Assume $s\sigma \sqsupset_{\mathsf{tcap}} t\tau$, so there exists a rule $\ell \to r \in \mathcal{R}$ such that $\mathsf{tcap}(s\sigma) \sim \ell$. Remember that the latter is just a shorthand for the existence of a substitution μ such that $\ell\mu \in [\![s\sigma]\!]$. By induction on u we have that $[\![\mathsf{tcap}(u\mu)]\!] \subseteq [\![\mathsf{tcap}(u)]\!]$ for any substitution μ. Hence also $\ell\mu \in [\![s]\!]$ and thus $\mathsf{tcap}(s) \sim \ell$. But then by definition $s \sqsupset_{\mathsf{tcap}} t$. Now assume $s \to^* t$ and $s \not\sqsupset_{\mathsf{tcap}} t$. From the latter we have that $\mathsf{tcap}(s) \not\sim \ell$ for all $\ell \to r \in \mathcal{R}$. If there would be any root step in $s \to^* t$ then $\mathsf{tcap}(s) \sim \ell$ for some rule $\ell \to r \in \mathcal{R}$. Hence $s \xrightarrow{>\epsilon}{}^* t$. So $\sqsupset_{\mathsf{tcap}}$ admits decomposition.

Next consider $\sqsupset_{\mathsf{rs}}$. Assume $s\sigma \sqsupset_{\mathsf{rs}} t\tau$. So by definition either $s\sigma \in \mathcal{V}$, $t\tau \in \mathcal{V}$, $\mathsf{root}(s\sigma) \sqsupset^+_{\mathsf{stg}} \mathsf{root}(t\tau)$, or $\mathsf{root}(s\sigma) \sqsupset^+_{\mathsf{stg}} \bot$. But then obviously $s \sqsupset_{\mathsf{rs}} t$ because for any term u and any substitution μ if $u\mu \in \mathcal{V}$ certainly also $u \in \mathcal{V}$ and if $u \notin \mathcal{V}$ then $\mathsf{root}(u\mu) = \mathsf{root}(u)$. Now assume $s \to^* t$ and $s \not\sqsupset_{\mathsf{rs}} t$. From the latter we have that neither s nor t are variables, as well as $\mathsf{root}(s) \not\sqsupset^+_{\mathsf{stg}} \mathsf{root}(t)$ and $\mathsf{root}(s) \not\sqsupset^+_{\mathsf{stg}} \bot$. But that means that there is no root step in $s \to^* t$ and hence $s \xrightarrow{>\epsilon}{}^* t$. So also $\sqsupset_{\mathsf{rs}}$ admits decomposition. □

Furthermore, if two relations admit decomposition then also their intersection does.

Lemma 7.23. *If $\sqsupset_1$ and $\sqsupset_2$ admit decomposition then so does $\sqsupset_1 \cap \sqsupset_2$.* □

So in our implementation we employ the intersection of the two relations defined earlier.

Corollary 7.24. *$\sqsupset_{tcap} \cap \sqsupset_{\mathsf{rs}}$ admits decomposition.* □

This is stronger than relying on the two relations separately as shown in the following example.

Example 7.25. Consider the TRS $\mathcal{R}_4$ from Example 7.21 above. Look at the two terms $s = \mathsf{f}(\mathsf{p}(\mathsf{a}))$ and $t = \mathsf{f}(\mathsf{b})$ and assume we want to know if $s\sigma \to^* t\tau$ for any substitutions σ and τ. We want to employ the abstract non-reachability check and the abstract decomposition function. In our first attempt we instantiate $\sqsupset$ with $\sqsupset_{\mathsf{rs}}$. Since the root symbols of s and t are the same $\mathsf{nonreach}(s,t)$ does not hold. We try to decompose the problem and get $\mathsf{D}_{s,t} = \{(\mathsf{p}(\mathsf{a}), \mathsf{b})\}$ because $s \not\sqsupset_{\mathsf{rs}} t$. Unfortunately $\mathsf{p}(\mathsf{a}) \sqsupset_{\mathsf{rs}} \mathsf{b}$ which means that $\mathsf{nonreach}(\mathsf{p}(\mathsf{a}), \mathsf{b})$ is not true. Clearly $\sqsupset_{\mathsf{rs}}$ does not work.

Let's try to instantiate $\sqsupset$ with $\sqsupset_{\mathsf{tcap}}$. Since $s \sqsupset_{\mathsf{tcap}} t$ neither $\mathsf{nonreach}(s,t)$ holds nor can we decompose the problem and we immediately have to give up.

So let's finally try to instantiate $\sqsupset$ with $\sqsupset_{\mathsf{tcap}} \cap \sqsupset_{\mathsf{rs}}$. The root symbols of s and t are the same so $\mathsf{nonreach}(s,t)$ does not hold. Since $s \not\sqsupset_{\mathsf{rs}} t$ also $s \; (\sqsupset_{\mathsf{tcap}} \cap \sqsupset_{\mathsf{rs}}) \; t$ does not hold and thus $\mathsf{D}_{s,t} = \{(\mathsf{p}(\mathsf{a}), \mathsf{b})\}$. We have $\mathsf{p}(\mathsf{a}) \not\sqsupset_{\mathsf{tcap}} \mathsf{b}$ and thus $\mathsf{nonreach}(\mathsf{p}(\mathsf{a}), \mathsf{b})$, which together with Lemma 7.19 yields $\mathsf{p}(\mathsf{a})\sigma \not\to^* \mathsf{b}\tau$. From this we get non-reachability of t from s by Lemma 7.17 and we are done.

7.4 Exact Tree Automata Completion

What is generally known as tree automata completion today was introduced by Genet. Jacquemard used a similar concept to show decidability of reachability for linear and growing TRSs. His proof was based on the construction of a tree automaton that accepts the set of ground terms which are normalizable with respect to a given linear and growing TRS $\mathcal{R}$. If we replace the automaton recognizing $\mathcal{R}$-normal forms in Jacquemard's construction by an arbitrary automaton $\mathcal{A}$ we arrive at a tree automaton that accepts the $\mathcal{R}$-ancestors of the language of $\mathcal{A}$.

We need some basic definitions and auxiliary lemmas before we present the construction of this *ancestor automaton* in detail.

Definition 7.26 (Ground-instances)**.** The set of *ground-instances* of a term t, that is, the set of terms s such that $s = t\sigma$ for some ground substitution σ is denoted by $\Sigma(t)$.

Definition 7.27 (Growingness, linear growing approximation)**.** A TRS $\mathcal{R}$ is called *growing* if for all $\ell \to r \in \mathcal{R}$ the variables in $\mathcal{V}(\ell) \cap \mathcal{V}(r)$ occur at depth at most one in ℓ. Given a TRS $\mathcal{R}$ the *linear growing approximation* is defined as any linear growing TRS obtained from $\mathcal{R}$ by linearizing the left-hand sides, renaming the variables in the right-hand sides that occur at a depth greater than one in the corresponding left-hand side, and finally also linearizing the right-hand sides (see Section 7.8). The linear growing approximation of a TRS $\mathcal{R}$ is denoted by $\mathsf{g}(\mathcal{R})$.

Definition 7.28 (Ground-instance transitions, Δ_t)**.** Let $[t]$ denote a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ where all variable-occurrences have been replaced by a fresh symbol $\Box$ (similar to Definitions 7.2 and 7.3). Using such terms as states we define the set Δ_t that contains all transitions which are needed to recognize all ground-instances of a term $t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ in state $[t]$.

$$\Delta_t = \begin{cases} \{f([t_1], \ldots, [t_n]) \to [t]\} \cup \bigcup_{1 \leqslant i \leqslant n} \Delta_{t_i} & \text{if } t = f(t_1, \ldots, t_n) \\ \{f(\Box, \ldots, \Box) \to \Box \mid f \in \mathcal{F}\} & \text{otherwise} \end{cases}$$

Note that if t is not linear this actually gives an overapproximation. The next lemma holds by definition of Δ_t.

Lemma 7.29. *For any subterm s of any term t if there is a sequence $u \to^+_{\Delta_t} [s]$ then u is a ground-instance of s, and vice versa if t is linear.* □

We now use Δ_t to define an automaton for the ground-instances of t.

Definition 7.30 (Ground-instance automaton, $\mathcal{A}_{\Sigma(t)}$)**.** Let Q_t denote the set of states occurring in Δ_t then we call the tree automaton $\mathcal{A}_{\Sigma(t)} = \langle \mathcal{F}, Q_t, \{[t]\}, \Delta_t \rangle$ the *ground-instance automaton for t*.

Lemma 7.31. *The language of $\mathcal{A}_{\Sigma(t)}$ is an overapproximation of the set of ground-instances of t in general and an exact characterization if t is linear.* □

Using the concept of ground-instance automaton we are now able to define a tree automaton which accepts all $\mathcal{R}$-ancestors of a given regular set of ground terms using exact tree automata completion.

Definition 7.32 (Ancestor automaton, $\mathsf{anc}_{\mathcal{R}}(\mathcal{A})$)**.** Given a tree automaton $\mathcal{A} = \langle \mathcal{F}, Q_{\mathcal{A}}, Q_f, \Delta \rangle$ whose states are all accessible, and a linear and growing TRS $\mathcal{R}$ the construction proceeds as follows.

First we extend the set of transitions of $\mathcal{A}$ in such a way that we can match left-hand sides of rules in $\mathcal{R}$. This yields the set of transitions $\Delta_0 = \Delta \cup \bigcup_{\ell \to r \in \mathcal{R}} \Delta_\ell$. Let $\mathcal{A}_0 = \langle \mathcal{F}, Q, Q_f, \Delta_0 \rangle$ where Q denotes the set of states in Δ_0. We have to ensure (for example by using the disjoint union of states) that for any state q which is used in both Δ and some Δ_ℓ, the terms which can reach it are the same ($\{t \mid t \to^+_{\Delta} q\} = \{t \mid t \to^+_{\Delta_\ell} q\}$). Then the language does not change, that is, $L(\mathcal{A}_0) = L(\mathcal{A})$.

Finally, we saturate Δ_0 by inference rule (†) in order to extend the language by $\mathcal{R}$-ancestors, that is, if we can reach a state q from an instance of a right-hand side of a rule in $\mathcal{R}$ we add a transition which ensures that q is reachable from the corresponding left-hand side:

$$\frac{f(\ell_1, \ldots, \ell_n) \to r \in \mathcal{R} \quad r\theta \to^*_{\Delta_k} q}{f(q_1, \ldots, q_n) \to q \in \Delta_{k+1}} \qquad (\dagger)$$

Here $\theta \colon \mathcal{V}(r) \to Q$ is a state substitution and $q_i = \ell_i\theta$ if ℓ_i is a variable in r and $q_i = [\ell_i]$ otherwise. Note that this inductive definition possibly adds many new transitions from Δ_k to Δ_{k+1}.

Since $\mathcal{R}$ is finite, the number of states is finite, and we do not introduce new states using (†), this process terminates after finitely many steps resulting in the set of transitions Δ_m. Also note that Δ_k is monotone with respect to k, that is, $\Delta_k \subseteq \Delta_{k+1}$ for all $k \geqslant 0$. We call $\mathsf{anc}_\mathcal{R}(\mathcal{A}) = \langle \mathcal{F}, Q, Q_f, \Delta_m \rangle$ the $\mathcal{R}$-ancestors automaton for $\mathcal{A}$. It is easy to show that $L(\mathcal{A}_0) \subseteq L(\mathsf{anc}_\mathcal{R}(\mathcal{A}))$.

Theorem 7.33. *Given a tree automaton $\mathcal{A}$ as well as a linear and growing TRS $\mathcal{R}$ the language of $\mathsf{anc}_\mathcal{R}(\mathcal{A})$ is exactly the set of $\mathcal{R}$-ancestors of $L(\mathcal{A})$.*

Proof. First we prove that $(\to_\mathcal{R}^*)[L(\mathcal{A})] \subseteq L(\mathsf{anc}_\mathcal{R}(\mathcal{A}))$. Pick a term $s \in (\to_\mathcal{R}^*)[L(\mathcal{A})]$. That means that there is a rewrite sequence $s \to_\mathcal{R}^k t$ of length $k \geqslant 0$ for some $t \in L(\mathcal{A})$. We proceed by induction on k. If $k = 0$ then $s = t$ and hence $s \in L(\mathsf{anc}_\mathcal{R}(\mathcal{A}))$. Now assume $k = k' + 1$ for some $k' \geqslant 0$ then there is a rewrite sequence $s = C[f(\ell_1, \ldots, \ell_n)\sigma] \to_\mathcal{R} C[r\sigma] \to_\mathcal{R}^{k'} t$ for some context C, rewrite rule $f(\ell_1, \ldots, \ell_n) \to r \in \mathcal{R}$, and substitution σ. By induction hypothesis $C[r\sigma] \in L(\mathsf{anc}_\mathcal{R}(\mathcal{A}))$. But that means that there is a state substitution $\theta\colon \mathcal{V}(r) \to Q$, a state $q \in Q$, and a final state $q_f \in Q_f$ such that $C[r\sigma] \to_{\Delta_m}^* C[r\theta] \to_{\Delta_m}^* C[q] \to_{\Delta_m}^* q_f$. From the construction using rule (†) we know that there is a transition $f(q_1, \ldots, q_n) \to q \in \Delta_m$ such that $q_i = \ell_i\theta$ if $\ell_i \in \mathcal{V}(r)$ and $q_i = [\ell_i]$ otherwise. If $\ell_i \in \mathcal{V}(r)$ then $\ell_i\sigma \to_{\Delta_m}^+ \ell_i\theta$ and otherwise $\ell_i\sigma \to_{\Delta_m}^+ [\ell_i]$ for all $1 \leqslant i \leqslant n$. Hence in both cases $\ell_i\sigma \to_{\Delta_m}^+ q_i$. But then we can construct the sequence $s = C[f(\ell_1\sigma, \ldots, \ell_n\sigma)] \to_{\Delta_m}^* C[f(q_1, \ldots, q_n)] \to_{\Delta_m} C[q] \to_{\Delta_m}^* q_f$ and hence $s \in L(\mathsf{anc}_\mathcal{R}(\mathcal{A}))$.

For the other direction we prove the following two properties for all sequences $s \to_{\Delta_m}^+ q$:

1. If $q = [t]$ for some subterm of a left-hand side of a rule in $\mathcal{R}$ then $s \in (\to_\mathcal{R}^*)[\Sigma(t)]$.

2. If $q \in Q_f$ then $s \in (\to_\mathcal{R}^*)[L(\mathcal{A})]$.

The proof for both properties works along the same lines. We sketch the one for the first property here. From the construction using rule (†) we know that $s \to_{\Delta_k}^+ [t]$ for some $k \geqslant 0$. We proceed by induction on k. If $k = 0$ then $s \to_{\Delta_0}^+ [t]$. By construction of $\mathcal{A}_0$ and Lemma 7.29 we have $s \in \Sigma(t)$ and hence also $s \in (\to_\mathcal{R}^*)[\Sigma(t)]$. Now assume that $k = k' + 1$ for some $k' \geqslant 0$. By induction hypothesis $s \to_{\Delta_{k'}}^+ [t]$ implies $s \in (\to_\mathcal{R}^*)[\Sigma(t)]$ for all terms s and t. Consider the set $\Delta_{k'+1} \setminus \Delta_{k'}$ of transitions which were newly added in $\Delta_{k'+1}$. We continue by a second induction on the size of $\Delta_{k'+1} \setminus \Delta_{k'}$. If it is empty we have a $\Delta_{k'}$-sequence and may simply close the proof with an application of the first induction hypothesis. Otherwise we have some set Δ and transition $\rho\colon f(q_1, \ldots, q_n) \to q'$ that was created from some rule $\ell \to r \in \mathcal{R}$ with $\ell = f(\ell_1, \ldots, \ell_n)$ and the sequence $r\theta \to_{\Delta_k'}^* q'$ by an application of rule (†) such that $\{\rho\} \uplus \Delta \subseteq \Delta_{k'+1} \setminus \Delta_{k'}$. The second induction hypothesis is if $s \to_{\Delta \cup \Delta_{k'}}^+ [t]$ then $s \in (\to_\mathcal{R}^*)[\Sigma(t)]$. Let m denote the number of steps that use ρ. We continue by a third induction on m. If $m = 0$ the sequence from s to $[t]$ only used transitions in $\Delta \cup \Delta_{k'}$ and using the second induction hypothesis we are done. Otherwise there is some $m' \geqslant 0$ such that $m = m' + 1$ and the induction hypothesis is that for all terms s, t if $s \to_{\Delta \cup \Delta_{k'}}^+ [t]$ using ρ only m' times then $s \in (\to_\mathcal{R}^*)[\Sigma(t)]$. Now we look at the first step using ρ in the sequence, that is, $s = D[f(s_1, \ldots, s_n)] \to_{\Delta \cup \Delta_{k'}}^* C[f(q_1, \ldots, q_n)] \to_\rho C[q'] \to_{\Delta_{k'+1}}^* [t]$. Note that from this we get $D[u] \to_{\Delta \cup \Delta_{k'}}^* C[u]$ for all terms u.

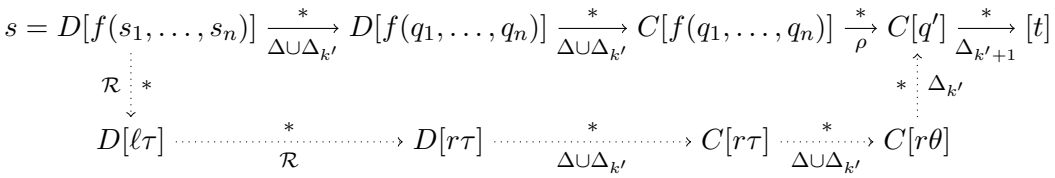


Figure 7.1: Bypassing ρ to close the induction step.

Next we define a substitution τ such that

$$s \to_{\mathcal{R}}^* D[\ell\tau] \to_{\mathcal{R}} D[r\tau] \to_{\Delta\cup\Delta_{k'}}^* C[r\tau] \to_{\Delta\cup\Delta_{k'}}^* C[q']$$

This allows us to bypass the ρ-step and so we arrive at a $\Delta_{k'+1}$-sequence from $D[r\tau]$ to $[t]$ containing one less ρ-step as shown in Figure 7.1. The construction of τ proceeds as follows: We fix $1 \leqslant i \leqslant n$. If ℓ_i is a variable in r define τ_i to be $\{\ell_i \mapsto s_i\}$. Otherwise we know from the definition of inference rule (†) that $q_i = [\ell_i]$ and $s_i \to_{\Delta\cup\Delta_{k'}}^+ [\ell_i]$. From that we have that $s_i \in (\to_{\mathcal{R}}^*)[\Sigma(\ell_i)]$ but that means that there is some substitution τ_i such that $s_i \to_{\mathcal{R}}^* \ell_i\tau_i$. Moreover let $\tau' = \{x \mapsto u_x \mid x \in \mathcal{V}(r) \setminus \mathcal{V}(\ell)\}$ where u_x is an arbitrary but fixed ground term such that $u_x \to_{\Delta_0}^* x\theta$. Since all states in $\mathcal{A}_0$ are accessible we can always find such a term u_x. Now let τ be the disjoint union of $\tau_1, \ldots, \tau_n, \tau'$. This substitution is well-defined because ℓ is linear. By construction of τ we get $s \to_{\mathcal{R}}^* D[\ell\tau]$.

Consider a variable x occurring in r. If x also occurs in ℓ we have $x = \ell_i$ for some unique $1 \leqslant i \leqslant n$ because $\mathcal{R}$ is growing. But then by construction of τ_i we get $x\tau = \ell_i\tau_i = s_i$. Moreover from the definition of q_i in inference rule (†) we have $q_i = \ell_i\theta = x\theta$. But then $x\tau \to_{\Delta\cup\Delta_{k'}}^+ x\theta$ from $s_i \to_{\Delta\cup\Delta_{k'}}^+ q_i$. On the other hand, if x does not occur in ℓ then $x\tau = x\tau'$ and $x\tau' \to_{\Delta_0}^* x\theta$ by construction of τ'. So in both cases $r\tau \to_{\Delta\cup\Delta_{k'}}^* r\theta$. Together with $r\theta \to_{\Delta_{k'}}^* q'$ and $C[q'] \to_{\Delta_{k'+1}}^* [t]$ we may construct the sequence $D[r\tau] \to_{\Delta_{k'+1}}^+ q_f$ which uses ρ only m' times. Using the induction hypothesis we arrive at $D[r\tau] \in (\to_{\mathcal{R}}^*)[\Sigma(t)]$. Together with $s \to_{\mathcal{R}}^* D[\ell\tau] \to_{\mathcal{R}}^* D[r\tau]$ this means that $s \in (\to_{\mathcal{R}}^*)[\Sigma(t)]$ and we are done. □

Lemma 7.34 (Non-reachability via anc)**.** *Let $\mathcal{R}$ be a linear and growing TRS over signature $\mathcal{F}$. We may conclude non-reachability of t from s if*

$$L(\mathcal{A}_{\Sigma(s)}) \cap L(\mathsf{anc}_{\mathcal{R}}(\mathcal{A}_{\Sigma(t)})) = \varnothing$$ □

Example 7.35 (Infeasibility via anc)**.** Consider the CTRS $\mathcal{R}$ consisting of the following five rules:

$$\mathsf{g}(\mathsf{a}, x) \to \mathsf{c} \Leftarrow \mathsf{f}(x, \mathsf{a}) \approx \mathsf{a} \qquad \mathsf{f}(\mathsf{a}, x) \to \mathsf{a} \qquad \mathsf{c} \to \mathsf{c}$$
$$\mathsf{g}(x, \mathsf{a}) \to \mathsf{d} \Leftarrow \mathsf{f}(x, \mathsf{b}) \approx \mathsf{b} \qquad \mathsf{f}(\mathsf{b}, x) \to \mathsf{b}$$

It has two critical pairs

$$\mathsf{c} \approx \mathsf{d} \Leftarrow \mathsf{f}(\mathsf{a}, \mathsf{b}) \approx \mathsf{b}, \mathsf{f}(\mathsf{a}, \mathsf{a}) = \mathsf{a}$$

and the symmetric one. Since $\mathsf{tcap}(\mathsf{f}(\mathsf{a},\mathsf{b})) = x \sim \mathsf{b}$ and $\mathsf{tcap}(\mathsf{f}(\mathsf{a},\mathsf{a})) = x \sim \mathsf{a}$ as well as $\mathsf{f} \sqsupset^{+}_{\mathsf{stg}} \mathsf{b}$ and $\mathsf{f} \sqsupset^{+}_{\mathsf{stg}} \mathsf{a}$ neither unification nor the symbol transition graph are sufficient to show infeasibility of these critical pairs. On the other hand, since the underlying TRS $\mathcal{R}_u$ is linear and growing, we may construct the tree automaton $\mathcal{A}_{\Sigma(\mathsf{f}(\mathsf{a},\mathsf{b}))}$ consisting of the three transitions

$$\mathsf{a} \to [\mathsf{a}] \qquad \mathsf{b} \to [\mathsf{b}] \qquad \mathsf{f}([\mathsf{a}],[\mathsf{b}]) \to [\mathsf{f}(\mathsf{a},\mathsf{b})]$$

where $[\mathsf{f}(\mathsf{a},\mathsf{b})]$ is the final state, as well as the tree automaton $\mathsf{anc}_{\mathcal{R}_u}(\mathcal{A}_{\Sigma(\mathsf{b})})$ consisting of 20 transitions

$$\begin{array}{lllll}
\mathsf{a} \to \Box & \mathsf{a} \to [\mathsf{a}] & \mathsf{g}(\Box,\Box) \to \Box & \mathsf{f}([\mathsf{b}],\Box) \to \Box & \mathsf{g}(\Box,[\mathsf{a}]) \to [\mathsf{g}(\Box,\mathsf{a})] \\
\mathsf{b} \to \Box & \mathsf{b} \to [\mathsf{b}] & \mathsf{g}(\Box,[\mathsf{a}]) \to \Box & \mathsf{f}([\mathsf{a}],\Box) \to [\mathsf{a}] & \mathsf{f}([\mathsf{a}],\Box) \to [\mathsf{f}(\mathsf{a},\Box)] \\
\mathsf{c} \to \Box & \mathsf{c} \to [\mathsf{c}] & \mathsf{g}([\mathsf{a}],\Box) \to \Box & \mathsf{f}([\mathsf{b}],\Box) \to [\mathsf{b}] & \mathsf{g}([\mathsf{a}],\Box) \to [\mathsf{g}(\mathsf{a},\Box)] \\
\mathsf{d} \to \Box & \mathsf{f}(\Box,\Box) \to \Box & \mathsf{f}([\mathsf{a}],\Box) \to \Box & \mathsf{g}([\mathsf{a}],\Box) \to [\mathsf{c}] & \mathsf{f}([\mathsf{b}],\Box) \to [\mathsf{f}(\mathsf{b},\Box)]
\end{array}$$

with final state $[\mathsf{b}]$. Because the language of the intersection automaton is empty we have shown infeasibility of the condition $\mathsf{f}(\mathsf{a},\mathsf{b}) \approx \mathsf{b}$ by Lemma 7.34. So both critical pairs are infeasible.

In the setting of Chapter 4 the right-hand sides of conditions are always linear terms (because of right-stability; see Definition 4.5). Hence it is beneficial to start with the ground-instance automaton for the right-hand side of a condition (which in this case is exact) and compute the set of ancestors rather than taking the possibly non-linear left-hand side of a condition, overapproximating the ground-instances and only then computing the descendants of this set. Although this is not necessarily true in the setting of Chapter 5 (there we have no linearity-restriction on the right-hand sides of conditions) we employ the same setup for the sake of convenience.

For one of our main use cases, Theorem 4.14, we are restricted to left-linear CTRSs (via almost orthogonality) and linear right-hand sides of conditions (via right-stability). The latter also holds for right-hand sides that are combined by a compound symbol (again by right-stability). We show that in this setting anc subsumes tcap (at least in theory and for the forward direction).

Lemma 7.36. *Let $\mathcal{R}$ be a left-linear CTRS and t a linear term. If tcap can show non-reachability of t from s, then so can anc.*

Proof. Below we write $\mathsf{ren}(t)$ for a linearization of the term t using fresh variables. We proof the contrapositive and assume that anc cannot show non-reachability. Moreover, let $\mathcal{R}'$ denote the result of applying the linear growing approximation to $\mathcal{R}_u$. Then there is some term u such that $u \in L(\mathcal{A}_{\Sigma(s)})$ and $u \in L(\mathsf{anc}_{\mathcal{R}'}(\mathcal{A}_{\Sigma(t)}))$. Since t is linear and $\mathcal{R}'$ is linear and growing the latter implies that $u \in (\to^{*}_{\mathcal{R}'})[\Sigma(t)]$ by Theorem 7.33 and thus $u \to^{*}_{\mathcal{R}'} t\tau$ for some substitution τ. By Lemma 7.4, this means that $t\tau \in [\![\mathsf{tcap}_{\mathcal{R}'}(u)]\!]$. Since $u \in L(\mathcal{A}_{\Sigma(s)})$, it is clearly the case that $u \in \Sigma(\mathsf{ren}(s))$ and thus $u = \mathsf{ren}(s)\sigma$ for some substitution σ. Moreover $[\![\mathsf{tcap}_{\mathcal{R}'}(u)]\!] \subseteq [\![\mathsf{tcap}_{\mathcal{R}'}(\mathsf{ren}(s))]\!] = [\![\mathsf{tcap}_{\mathcal{R}'}(s)]\!]$. Together

with $t\tau \in [\![\mathsf{tcap}_{\mathcal{R}'}(u)]\!]$ from above, we obtain $t\tau \in [\![\mathsf{tcap}_{\mathcal{R}'}(s)]\!]$. However, tcap does only consider the left-hand sides of rules, which are the same in $\mathcal{R}'$ and $\mathcal{R}_u$, thus also $t\tau \in [\![\mathsf{tcap}_{\mathcal{R}_u}(s)]\!]$ which implies $\mathsf{tcap}_{\mathcal{R}_u}(s) \sim t$. □

If we also consider the reverse direction, that is, checking if the term s is reachable from t by $\mathcal{R}_u^{-1}$ for some condition $s \approx t$ in Theorem 4.14, then tcap may well succeed where anc fails, as shown by the next example.

Example 7.37 (anc vs. tcap)**.** The oriented 3-CTRS $\mathcal{R}$ consisting of the two rules

$$\mathsf{g}(x) \to \mathsf{g}(x) \Leftarrow \mathsf{g}(x) \approx \mathsf{f}(\mathsf{a},\mathsf{b}) \qquad\qquad \mathsf{g}(x) \to \mathsf{f}(x,x)$$

is right-stable and extended properly oriented. It has two symmetric CCPs of the form

$$\mathsf{f}(x,x) \approx \mathsf{g}(x) \Leftarrow \mathsf{g}(x) \approx \mathsf{f}(\mathsf{a},\mathsf{b}).$$

The underlying TRS $\mathcal{R}_u$ is not linear and growing, so if we want to apply anc we have to apply a linear growing approximation, resulting for example in $\mathcal{R}'$

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

But then anc is not able to show infeasibility since the language accepted by the tree automaton $\mathcal{A}_{\Sigma(\mathsf{g}(x))} \cap \mathsf{anc}_{\mathcal{R}'}(\mathcal{A}_{\Sigma(\mathsf{f}(\mathsf{a},\mathsf{b}))})$ is not empty and also for the reverse direction $\mathcal{A}_{\Sigma(\mathsf{f}(\mathsf{a},\mathsf{b}))} \cap \mathsf{anc}_{\mathcal{R}'^{-1}}(\mathcal{A}_{\Sigma(\mathsf{g}(x))})$ we get a non-empty language. On the other hand, using the reversed underlying system $\mathcal{R}_u^{-1}$

$$\mathsf{f}(x,x) \to \mathsf{g}(x) \qquad\qquad \mathsf{g}(x) \to \mathsf{g}(x)$$

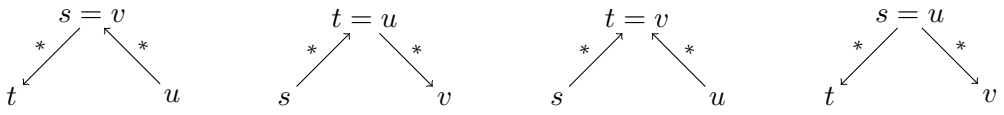
we have that $\mathsf{tcap}_{\mathcal{R}_u^{-1}}(\mathsf{f}(\mathsf{a},\mathsf{b})) = \mathsf{f}(\mathsf{a},\mathsf{b}) \not\sim \mathsf{g}(x)$. So in this case tcap succeeds where anc fails.

7.5 Equational Reasoning

In this section we want to briefly look at another method that was also first used for computing dependency graphs. It employs Waldmeister, a powerful automatic theorem prover for equational logic with uninterpreted function symbols. Waldmeister uses a variant of ordered completion to determine for a given set of equations $\mathcal{R}$ and a goal equation (called conclusion) $s \approx t$ whether there exist substitutions σ and τ such that $s\sigma \leftrightarrow_{\mathcal{R}}^* t\tau$. If Waldmeister refutes the conclusion then surely there are no substitutions σ and τ such that $s\sigma \to_{\mathcal{R}}^* t\tau$.

Example 7.38. Consider the CTRS $\mathcal{R}$ consisting of the following nine rules:

$$\begin{array}{ccc}
0 \leqslant x \to \mathsf{true} & \mathsf{s}(x) > 0 \to \mathsf{true} & x - 0 \to x \\
\mathsf{s}(x) \leqslant \mathsf{s}(y) \to x \leqslant y & \mathsf{s}(x) > \mathsf{s}(y) \to x > y & 0 - x \to 0 \\
x \div y \to \langle 0, y\rangle \quad \Leftarrow y > x \approx \mathsf{true} & & \mathsf{s}(x) - \mathsf{s}(y) \to x - y \\
x \div y \to \langle \mathsf{s}(q), r\rangle \Leftarrow y \leqslant x \approx \mathsf{true},\ (x-y) \div y \approx \langle q, r\rangle & &
\end{array}$$



(a) $u \to^* t$ (b) $s \to^* v$ (c) $s \downarrow u$ (d) $t \downarrow v$

Figure 7.2: Requirements for the conditions c containing $s \approx t$ and $u \approx v$ to be satisfiable.

This CTRS has two trivial unconditional CPs and one (modulo symmetry) CCP

$$\langle 0, x\rangle \approx \langle \mathsf{s}(y), z\rangle \Leftarrow x \leqslant w \approx \mathsf{true},\ (w - x) \div x \approx \langle y, z\rangle,\ x > w \approx \mathsf{true}$$

which is infeasible because of the contradictory conditions $x \leqslant w \approx \mathsf{true}$ and $x > w \approx \mathsf{true}$. This is confirmed by Waldmeister in conjunction with the $\mathcal{R} \mapsto \mathcal{R}_{\mathsf{u}}$ transformation.

Because the output of Waldmeister does not give enough details to certify it, this is the only infeasibility method used in ConCon that is not certifiable and hence only available in the uncertified mode.

7.6 Exploiting Equalities

Finally, there are four cases where the equality between some terms in the conditions can be exploited to check a CCP for infeasibility. These four situations are depicted in Figure 7.2 and explained below.

Assume we have a CCP $\ell \approx r \Leftarrow c$ where c contains at least two different conditions $s \approx t$ and $u \approx v$. Now $s = v$ implies that for c to be satisfiable t has to be reachable from u (see Figure 7.2(a)). So if we can show that t is not reachable from u we know that the conditions c are infeasible. Similarly for $t = u$ to have any chance to satisfy the conditions v has to be reachable from s otherwise c is infeasible (see Figure 7.2(b)). On the other hand if $t = v$ then to satisfy the conditions s and u have to be joinable and so from non-joinability of s and u we can conclude infeasibility of c (see Figure 7.2(c)). Finally, if we are in the setting described in Chapter 4 then if $s = u$ to satisfy the conditions there would have to exist a diverging situation (see Figure 7.2(d)) but by the assumption of Definition 4.12.**3** this implies that there is a join between t and v so to proof infeasibility of c it suffices to show non-joinability of t and v.

7.7 Certification

In this section we give an overview of all infeasibility and non-reachability techniques that are supported by our certifier CeTA and what kind of information it requires from a certificate in CPF [56] (short for *certification problem format*). Before we come to the special infeasibility condition of Definition 4.12, we handle the common case where, given a list of conditions c, we are interested in proving $\sigma, n \not\vdash c$ for every substitution σ and level n.

Lemma 7.39 (Infeasibility certificates)**.** *Given $(\mathcal{R}, c)$ consisting of a CTRS $\mathcal{R}$ and a list of conditions $c = s_1 \approx t_1, \ldots, s_k \approx t_k$, infeasibility of c with respect to $\mathcal{R}$ can be certified in one of the following ways:*

1. *Provide two terms s and t with $s \approx t \in c$, and a non-reachability certificate for $(\mathcal{R}_u, s, t)$.*
2. *Provide a function symbol cs of arity n together with a non-reachability certificate for $(\mathcal{R}_u, \mathsf{cs}(s_1, \ldots, s_k), \mathsf{cs}(t_1, \ldots, t_k))$.*
3. *For an arbitrary subset c' of c, provide an infeasibility certificate for $(\mathcal{R}, c')$.*
4. *Provide three terms s, t, and u such that $s \approx u$ and $t \approx u$ are equations in c together with a non-joinability certificate for $(\mathcal{R}_u, s, t)$.*

Proof. Note that **3** is obvious and **1** only exists for tool-author convenience but is subsumed by the combination of **2** and **3**. Moreover, **2** follows from the fact that $\mathsf{cs}(s_1, \ldots, s_k)\sigma \not\to^*_{\mathcal{R}_u} \mathsf{cs}(t_1, \ldots, t_k)\tau$ for all σ and τ, implies the existence of at least one $1 \leqslant i \leqslant k$ such that $s_i\sigma \not\to^*_{\mathcal{R}_u} t_i\tau$ for all σ and τ. Finally, for **4**, whenever $s\sigma$ and $t\tau$ are not joinable for arbitrary σ and τ, the existence of μ and n such that $\mu, n \vdash s \approx u, t \approx u$ is impossible. □

Note that in **2** we check for non-reachability between left-hand sides and their corresponding right-hand sides, while in **4** we check for non-joinability between two left-hand sides. Thus, while in general non-joinability is more difficult to show than non-reachability, **4** is not directly subsumed by **2**.

Lemma 7.40 (Non-reachability certificates)**.** *Given $(\mathcal{R}, s, t)$ consisting of a TRS $\mathcal{R}$ and two terms s and t, $\mathcal{R}$-non-reachability of t from s can be certified in one of the following ways:*

1. *Indicate that $\mathsf{tcap}(s)$ does not unify with t.*
2. *Provide a TRS $\mathcal{R}'$ such that for each $\ell \to r \in \mathcal{R}$ there is $\ell' \to r' \in \mathcal{R}'$ and a substitution σ with $\ell = \ell'\sigma$ and $r = r'\sigma$, together with a non-reachability certificate for $(\mathcal{R}', s, t)$.*
3. *Provide a non-reachability certificate for $(\mathcal{R}^{-1}, t, s)$.*
4. *Make sure that $\mathcal{R}$ is linear and growing and provide a finite signature $\mathcal{F}$ and two constants a and $\Box$ such that $\mathsf{a} \in \mathcal{F}$ but $\Box \notin \mathcal{F}$, together with a tree automaton $\mathcal{A}$ that is an overapproximation of $\mathsf{anc}_{\mathcal{R}}(\mathcal{A}_{\Sigma(t)})$ and satisfies $L(\mathcal{A}_{\Sigma(s)}) \cap L(\mathcal{A}) = \varnothing$.*

Proof. If $\mathsf{tcap}_{\mathcal{R}}(s) \not\sim t$, then **1** holds by Lemma 7.4. Further note that $\to_{\mathcal{R}} \subseteq \to_{\mathcal{R}'}$ and thus **2** is immediate. Moreover, **3** is obvious, leaving us with **4**. From a certification perspective this is the most interesting case. To begin with, there are two reasons why we do not want to repeat the full construction of anc inside CeTA. Firstly, we would unnecessarily repeat an operation with at least exponential worst-case complexity. Secondly, a fully-verified executable algorithm is not even part of our formalization, instead we heavily rely on inductive definitions. While turning the existing inductive definitions into executable recursive functions would definitely be possible, we stress that this is not necessary. In CeTA we check that $\mathcal{A}$ is an overapproximation of $\mathsf{anc}_{\mathcal{R}}(\mathcal{A}_{\Sigma(t)})$ as follows: firstly, we ensure that $\mathcal{A}$ does not contain epsilon transitions, that $[t]$ is included

in the final states of $\mathcal{A}$, and that Δ_t as well as the matching rules with respect to the signature $\mathcal{F}$ are part of the transitions of $\mathcal{A}$; secondly, we check that $\mathcal{A}$ is closed with respect to inference rule (†). Since $\mathcal{A}_{\Sigma(s)}$ is an overapproximation of $\Sigma(s)$ and by the required conditions together with Theorem 7.33, $L(\mathcal{A})$ overapproximates $[\to_{\mathcal{R}}^*](\Sigma(t))$, we can conclude $\Sigma(s) \cap [\to_{\mathcal{R}}^*](\Sigma(t)) = \varnothing$. Thus there are no *ground* substitutions σ and τ such that $s\sigma, t\tau \in \mathcal{T}(\mathcal{F})$ and $s\sigma \to_{\mathcal{R}}^* t\tau$. In order to conclude that the same holds true for *arbitrary* substitutions (not necessarily restricted to $\mathcal{F}$), we rely on an earlier result [55] that implies that whenever $s\sigma \to_{\mathcal{R}}^* t\tau$ for arbitrary σ and τ and $s, t \in \mathcal{T}(\mathcal{F}, \mathcal{V})$ then there are σ' and τ' such that $s\sigma', t\tau' \in \mathcal{T}(\mathcal{F})$ and $s\sigma' \to_{\mathcal{R}}^* t\tau'$. □

Note that **2** allows us to certify the linear growing approximation of a TRS without actually having to formalize it in Isabelle/HOL. More specifically, whenever $\mathcal{R}'$ is the result of applying the linear growing approximation to $\mathcal{R}$, then the corresponding certificate will pass **2** and $\mathcal{R}'$ will be linear and growing in the check for **4**; otherwise **4** will fail.

Lemma 7.41 (Non-joinability certificates)**.** *Given $(\mathcal{R}, s, t)$ consisting of a TRS $\mathcal{R}$ and two term s and t, $\mathcal{R}$-non-joinability of s and t can be certified in one of the following ways:*

1. *Indicate that $\mathsf{tcap}(s)$ does not unify with $\mathsf{tcap}(t)$.*
2. *If at least one of the terms, say t, is a ground $\mathcal{R}$-normal form provide a non-reachability certificate for $(\mathcal{R}, s, t)$.*

Proof. We prove **1** by Lemma 7.5 and **2** by Lemma 7.4 since non-joinability reduces to non-reachability when one of the terms is an $\mathcal{R}$-normal form. □

Lemma 7.42 (Ao-infeasibility certificates)**.** *Given $(\mathcal{R}, c_1, c_2)$ consisting of a CTRS $\mathcal{R}$ fulfilling all syntactic requirements of Theorem 4.14 and two lists of conditions c_1 and c_2, infeasibility with respect to almost orthogonality can be certified in one of the following ways:*

1. *Provide an infeasibility certificate for $(\mathcal{R}, c)$ where c is the concatenation of c_1 and c_2.*
2. *Provide three terms s, t and u such that $s \approx t$ is an equation in c_1 and $s \approx u$ an equation in c_2, together with a non-joinability certificate for $(\mathcal{R}_u, t, u)$.*

Proof. While **1** follows from Lemma 7.39, in **2** we make use of the level-commutation assumption of Definition 4.12 to deduce non-meetability of t and u from non-joinability of t and u. □

7.8 Chapter Notes

We have seen several techniques to proof infeasibility of CCPs in this chapter. Most of them are inspired by techniques for dependency graph analysis and other reachability problems. All but one of the methods are formalized in IsaFoR and hence certifiable by CeTA.

For details on the tcap function see [22]. Because typical definitions of tcap rely on replacing subterms by "fresh variables" they are somewhat hard to formalize as already remarked in [65]. For that reason we instead opted for the definition used in this chapter. Ground context matching has been shown to be decidable by an efficient algorithm by Thiemann and Christian Sternagel [65].

While the symbol transition graph was introduced by Yamada (personal communication), decomposition of reachability problems is due to Christian Sternagel. The formalization of these concepts in IsaFoR was also done by Yamada and Christian Sternagel.

Already in 1996 Jacquemard [31] used a concept similar to tree automata completion and he also introduced the notion of growingness of a TRS and the linear growing approximation [31]. Note that the definition of the linear growing approximation in this chapter is ambiguous, because the second step of Definition 7.27 depends on the first step and we have a choice of how to linearize the variables (see also [42]). Our exact tree automata completion method is based on Jacquemard's work. Adding the transitions in (†) is symmetric to resolving compatibility violations in the tree automata completion by Genet [19, 20].

Tree automata completion for sets of descendants was introduced by Genet in 1998 [19, 20]. In contrast to Jacquemard's procedure this one does not guarantee termination but instead is parameterized by an abstraction function which limits the number of newly generated states during completion, thereby providing a trade-off between the termination behavior (and thus runtime) of the process and its accuracy. It takes a tree automaton representing a regular set and a left-linear TRS as input. Compatibility violations between the tree automaton and the TRS are resolved in an iterative process and termination depends on the employed abstraction function. Later, in [15] Feuillade and Genet introduced conditional tree automata completion that directly operates on CTRSs. They showed that this direct approach results in smaller tree automata (thereby reducing the possibility of divergence of the completion process) compared to tree automata completion applied to $\mathcal{R}_{\mathsf{u}}$.

During development of ConCon (before even thinking about certification) we implemented all three tree automata techniques: the exact tree automata completion by Jacquemard, as well as both the unconditional tree automata completion and also its extension to the conditional case by Feuillade and Genet. Moreover, we adapted the latter, which was originally defined for join 1-CTRSs with at most one condition per rule, to oriented 3-CTRSs with an arbitrary number of conditions (see [59, Section 4]). In our experiments exact tree automata completion, however, outperformed (and even subsumed) the results we could get by the other two procedures (see [59, Section 6]). Because of this and because formalization of exact tree automata completion seemed more feasible than formalizing (conditional) tree automata completion ConCon today only features the procedure by Jacquemard.

Equational reasoning for computing dependency graphs was presented in [69]. For more details on Waldmeister see [17]. Parts of this chapter are based on our publications [53,59]. The CTRS in Example 7.1 is from [48, Example 6.7] (326), the one in Example 7.35 is from [53, Example 16] (494), the one in Example 7.37 is from [53, Example 23], and

finally the one in Example 7.38 is from [51, Example 9] (`361`).

The formalization of the methods described in this chapter can be found in the following IsaFoR theory files:

```
thys/Conditional_Rewriting/Infeasibility.thy
thys/Nonreachability/Nonreachability.thy
thys/Nonreachability/Gtcap.thy
thys/Tree_Automata/Exact_Tree_Automata_Completion.thy
thys/Tree_Automata/Exact_Tree_Automata_Completion_Impl.thy
```

In the next chapter we turn our attention to non-confluence. More specifically, we present simple methods to find witnesses that establish non-confluence.

Chapter 8

Non-Confluence

When checking CTRSs for confluence we are not only interested in positive answers. If we cannot show confluence of a CTRS the reason may very well be that it just is *not* confluent. In what follows we present three methods to check CTRSs for non-confluence. The first check only works on 4-CTRSs, the second takes unconditional critical pairs into account, and the last employs conditional narrowing to find non-confluence witnesses. The implementation of the checks in ConCon is straightforward. Since most of them are rather fast we employ them in parallel to checking for confluence. Finally, we look at how to certify the methods using CeTA.

8.1 Finding Non-Confluence Witnesses

To prove non-confluence of a CTRS we have to find a witness, that is, a situation in which there are two diverging rewrite sequences starting at the same term where the end points are not joinable.

The first criterion only works for CTRSs that contain at least one unconditional rule of type 4, that is, with extra-variables in the right-hand side.

Lemma 8.1. *Given a 4-CTRS $\mathcal{R}$ and an unconditional rule $\rho\colon \ell \to r$ in $\mathcal{R}$ where $\mathcal{V}(r) \not\subseteq \mathcal{V}(\ell)$ and r is a normal form with respect to $\mathcal{R}_{\mathsf{u}}$ then $\mathcal{R}$ is non-confluent.*

Proof. Since $\mathcal{V}(r) \not\subseteq \mathcal{V}(\ell)$ we can always find two renamings μ_1 and μ_2 restricted to $\mathcal{V}(r) \setminus \mathcal{V}(\ell)$ such that $r\mu_1 \ {}_{\rho}\!\leftarrow \ell\mu_1 = \ell\mu_2 \to_{\rho} r\mu_2$ and $r\mu_1 \neq r\mu_2$. As r is a normal form with respect to $\mathcal{R}_{\mathsf{u}}$ also $r\mu_1$ and $r\mu_2$ are (different) normal forms with respect to $\mathcal{R}_{\mathsf{u}}$ (and hence also with respect to $\mathcal{R}$). Because we found a non-joinable peak $\mathcal{R}$ is non-confluent. □

Example 8.2. Consider the 4-CTRS $\mathcal{R}$ consisting of the two rules

$$\mathsf{e} \to \mathsf{f}(x) \Leftarrow \mathsf{l} \approx \mathsf{d} \qquad\qquad \mathsf{A} \to \mathsf{h}(x, x)$$

The right-hand side $\mathsf{h}(x, x)$ of the second (unconditional) rule is a normal form with respect to the underlying TRS $\mathcal{R}_{\mathsf{u}}$ and the only variable occurring in it does not appear in its left-hand side A. So by Lemma 8.1 $\mathcal{R}$ is non-confluent.

A natural candidate for diverging situations are the critical peaks of a CTRS. We will base our next criterion on the analysis of *unconditional* critical pairs (CPs) of CTRSs.

This restriction is necessary to guarantee the existence of the actual peak. If we would also allow conditional CPs, we first would have to check for infeasibility, since infeasibility is undecidable in general these checks are potentially very costly (see Chapter 7).

Lemma 8.3. *Given a CTRS $\mathcal{R}$ and an unconditional CP $s \approx t$ of it. If s and t are not joinable with respect to $\mathcal{R}_\mathsf{u}$ then $\mathcal{R}$ is non-confluent.*

Proof. The CP $s \approx t$ originates from a critical overlap between two unconditional rules $\rho_1 \colon \ell_1 \to r_1$ and $\rho_2 \colon \ell_2 \to r_2$ for some mgu μ of $\ell_1|_p$ and ℓ_2 such that

$$s = \ell_1\mu[r_2\mu]_p \leftarrow \ell_1\mu[\ell_2\mu]_p \to r_1\mu = t.$$

Since s and t are not joinable with respect to $\mathcal{R}_\mathsf{u}$ they are of course also not joinable with respect to $\mathcal{R}$ and we have found a non-joinable peak. So $\mathcal{R}$ is non-confluent. □

Example 8.4. Consider the 3-CTRS $\mathcal{R}$ consisting of the four rules

$$\mathsf{p}(\mathsf{q}(x)) \to \mathsf{p}(\mathsf{r}(x)) \qquad \mathsf{q}(\mathsf{h}(x)) \to \mathsf{r}(x) \qquad \mathsf{r}(x) \to \mathsf{r}(\mathsf{h}(x)) \Leftarrow \mathsf{s}(x) \approx 0 \qquad \mathsf{s}(x) \to 1$$

First of all we can immediately drop the third rule because we can never satisfy its condition and so it does not influence the rewrite relation of $\mathcal{R}$ (see Section 9.1). This results in the TRS $\mathcal{R}'$. Now the left- and right-hand sides of the unconditional CP $\mathsf{p}(\mathsf{r}(z)) \approx \mathsf{p}(\mathsf{r}(\mathsf{h}(z)))$ are not joinable because they are two different normal forms with respect to the underlying TRS $\mathcal{R}'_\mathsf{u}$. Hence $\mathcal{R}$ is not confluent by Lemma 8.3.

While the above lemmas are easy to check and we have fast methods to do so they are also rather ad hoc. A more general but potentially very expensive way to search for non-joinable forks is to use narrowing. Narrowing is a generalization of rewriting where instead of matching we use unification.

Definition 8.5 (Narrowing)**.** Given a TRS $\mathcal{R}$ we say that s *narrows* to t, written $s \rightsquigarrow_\sigma t$ if there is a variant of a rule $\rho \colon \ell \to r \in \mathcal{R}$, such that $\mathcal{V}(s) \cap \mathcal{V}(\rho) = \varnothing$, a position $p \in \mathcal{P}\mathsf{os}_\mathcal{F}(s)$, an mgu σ of $s|_p$ and ℓ, and $t = s[r]_p\sigma$.

For a narrowing sequence $s_1 \rightsquigarrow_{\sigma_1} s_2 \rightsquigarrow_{\sigma_2} \cdots \rightsquigarrow_{\sigma_{n-1}} s_n$ of length n we write $s_1 \rightsquigarrow^n_\sigma s_n$ where $\sigma = \sigma_1\sigma_2\cdots\sigma_{n-1}$. If we are not interested in the length we also write $s \rightsquigarrow^*_\sigma t$. In an implementation we have to be careful to respect the freshness requirement of the variables in the used rule for every step with respect to all the previous terms and rules. Especially the following well-known property of narrowing will be useful:

Property 8.6. If $s \rightsquigarrow_\sigma t$ then $s\sigma \to t\sigma$ with the same rule that was employed in the narrowing step. Moreover, if we have $s_1 \rightsquigarrow_{\sigma_1} s_2 \rightsquigarrow_{\sigma_2} \cdots \rightsquigarrow_{\sigma_{n-1}} s_n$ then also $s_1\sigma_1\sigma_2\cdots\sigma_{n-1} \to s_2\sigma_2\cdots\sigma_{n-1} \to \cdots \to s_n$. Again employing the same rule for each rewrite step as in the corresponding narrowing step.

We can extend narrowing to the conditional case by simply taking the conditions into consideration.

Definition 8.7 (Conditional narrowing)**.** Given a CTRS $\mathcal{R}$ we say that s *conditionally narrows* to t, written $s \rightsquigarrow_\sigma t$ if there is a variant of a rule $\rho\colon \ell \to r \Leftarrow c \in \mathcal{R}$, such that $\mathcal{V}(s) \cap \mathcal{V}(\rho) = \varnothing$ and $u \rightsquigarrow_\sigma^* v$ for all $u \approx v \in c$, a position $p \in \mathcal{P}\mathsf{os}_\mathcal{F}(s)$, a unifier[1] σ of $s|_p$ and ℓ, and $t = s[r]_p\sigma$.

Using conditional narrowing we can now formulate a more general non-confluence criterion.

Lemma 8.8. *Given a CTRS $\mathcal{R}$, if we can find two narrowing sequences $u \rightsquigarrow_\sigma^* s$ and $v \rightsquigarrow_\tau^* t$ such that $u\sigma\mu = v\tau\mu$ for some mgu μ and $s\sigma\mu$ and $t\tau\mu$ are not joinable with respect to $\mathcal{R}_\mathsf{u}$ then $\mathcal{R}$ is non-confluent.*

Proof. Employing Property 8.6 we get the two rewriting sequences $u\sigma \to_\mathcal{R}^* s\sigma$ and $v\tau \to_\mathcal{R}^* t\tau$. Since rewriting is closed under substitutions we have the diverging situation $s\sigma\mu \,{}_\mathcal{R}^*\!\leftarrow u\sigma\mu = v\tau\mu \to_\mathcal{R}^* t\tau\mu$. As the two endpoints of these forking sequences $s\sigma\mu$ and $t\tau\mu$ are not joinable by $\mathcal{R}_\mathsf{u}$ they are certainly also not joinable by $\mathcal{R}$. This establishes non-confluence of the CTRS $\mathcal{R}$. □

Example 8.9. Remember the 3-CTRS from Example 5.3 consisting of the three rules

$$0 + y \to y \qquad \mathsf{s}(x) + y \to x + \mathsf{s}(y) \qquad \mathsf{f}(x, y) \to z \Leftarrow x + y \approx z + v$$

Starting from a variant of the left-hand side of the third rule $u = \mathsf{f}(x', y')$ we have a narrowing sequence $\mathsf{f}(x', y') \rightsquigarrow_\sigma x_1$ using the variant $\mathsf{f}(x_1, x_2) \to x_3 \Leftarrow x_1 + x_2 \approx x_3 + x_4$ of the third rule and the substitution $\sigma = \{x' \mapsto x_1, x_3 \mapsto x_1, x_4 \mapsto x_2\}$. We also have another narrowing sequence $\mathsf{f}(x', y') \rightsquigarrow_\tau x_3$ using the same variant of rule three and substitution $\tau = \{x \mapsto x_3 + x_4, x' \mapsto 0, y' \mapsto x_3 + x_4, x_1 \mapsto 0, x_2 \mapsto x_3 + x_4\}$ where for the condition $x_1 + x_2 \approx x_3 + x_4$ we have the narrowing sequence $x_1 + x_2 \rightsquigarrow_\tau x_3 + x_4$, using a variant of the first rule $0 + x \to x$. Finally, there is an mgu $\mu = \{x_1 \mapsto 0, x_2 \mapsto x_3 + x_4\}$ such that $u\sigma\mu = \mathsf{f}(0, x_3 + x_4) = u\tau\mu$. Moreover, $x_1\sigma\mu = 0$ and $x_3\tau\mu = x_3$ are two different normal forms. Hence $\mathcal{R}$ is non-confluent by Lemma 8.8.

8.2 Implementation

In ConCon our implementation of Lemma 8.1 takes the unconditional rule $\rho\colon \ell \to r$, the substitution $\sigma = \{x \mapsto y\}$ where $x \in \mathcal{V}(r) \setminus \mathcal{V}(\ell)$ and y is fresh with respect to ρ and builds the non-joinable fork $r \,{}_\rho\!\leftarrow \ell \to_\rho r\sigma$.

For Lemma 8.3 we have three concrete implementations that consider an overlap from which an unconditional CP $s \approx t$ arises:

1. The first of which just takes this overlap and then checks that s and t are two different normal forms with respect to $\mathcal{R}_\mathsf{u}$.
2. The second employs the tcap-function to check for non-joinability, that is, it checks whether $\mathsf{tcap}(s)$ and $\mathsf{tcap}(t)$ are not unifiable.

[1] In our implementation we actually start with an mgu of $s|_p$ and ℓ and then extend it while trying to satisfy the conditions.

3. The third makes a special call to the TRS confluence checker CSI providing the underlying TRS $\mathcal{R}_{\mathsf{u}}$ as well as the unconditional CP $s \approx t$ where all variables in s and t have been replaced by fresh constants. We issue the following command:

   ```
   csi -s '(nonconfluence -nonjoinability -steps 0 -tree)[30]' -C RT
   ```

   The strategy '`(nonconfluence -nonjoinability -steps 0 -tree)[30]`' tells CSI to check non-joinability of two terms using tree automata techniques. Here '`-steps 0`' means that CSI does not rewrite the input terms further before checking non-joinability (this would be unsound in our setting). The timeout is set to 30 seconds. To encode the two terms for which we want to check non-joinability in the input we set CSI to read relative-rewriting input ('`-C RT`'). We provide $\mathcal{R}_{\mathsf{u}}$ in the usual Cops-format and add one line for the CP $s \approx t$ where its "grounded" left- and right-hand sides are related by '`->=`', that is, we encode it as a relative rule. This is necessary to distinguish the unconditional CP from the rewrite rules.

Finally, for an implementation of Lemma 8.8 we have to be careful to respect the freshness requirement of the variables in the used rule for every narrowing step with respect to all the previous terms and rules. The crucial point is to efficiently find the two narrowing sequences, to this end we first restrict the set of terms from which to start narrowing. As a heuristic we only consider the left-hand sides of rules of the CTRS under consideration. Next we also prune the search space for narrowing. Here we restrict the length of the narrowing sequences to at most three. In experiments on Cops allowing sequences of length four or more did not yield additional non-confluence proofs but slowed down the tool significantly to the point where we lost other proofs. Further, we also limit the recursion depth of conditional narrowing by restricting the level (see Definition 2.42) to at most two. Again, we set this limit as tradeoff after thorough experiments on Cops. Ultimately, we use Property 8.6 to translate the forking narrowing sequences into forking conditional rewriting sequences. In this way we generate a lot of forking sequences so we only use fast methods, like non-unifiability of the tcap's of the endpoints or that they are different normal forms, to check for non-joinability of the endpoints. Calls to CSI are too expensive in this context.

8.3 Certification

Certification is quite similar for all of the methods described in this chapter. We have to provide a non-confluence witness, that is, a non-joinable fork: $t \,{}^{+}_{\mathcal{R}}\!\leftarrow s \rightarrow^{+}_{\mathcal{R}} u$. So besides the CTRS $\mathcal{R}$ under investigation we also need to provide the starting term s, the two endpoints of the fork t and u, as well as, certificates for $s \rightarrow^{+}_{\mathcal{R}} t$ and $s \rightarrow^{+}_{\mathcal{R}} u$, and a certificate that t and u are not joinable. For the forking rewrite sequences we reuse the formalization of conditional rewrite sequences described in Chapter 5 to build the certificates. We also want to stress that because of Property 8.6 we did not have to formalize conditional narrowing because going from narrowing to rewrite sequences is already done in ConCon and in the certificate only the rewrite sequences show up. For the non-joinability certificate of t and u there are three options: either we state that

t and u are two different normal forms or that $\mathsf{tcap}(t)$ and $\mathsf{tcap}(u)$ are not unifiable; both of these checks are performed within CeTA; or, when the witness was found by an external call to CSI, we just include the generated non-joinability certificate.

8.4 Chapter Notes

In this chapter we have seen three simple methods to check CTRSs for non-confluence.

To be closer to the presentation of conditional rewriting used in this thesis our definition of conditional narrowing deviates from the one commonly found in the literature, see [43], in that we do not define it on goal clauses but rather directly on terms.

Starting from its first participation in CoCo 2014, ConCon 1.2.0.3 came equipped with some non-confluence heuristics. Back then it only used Lemmas 8.1 and 8.3 and had no support for certification of the output. In the next two years (ConCon 1.3.0 and 1.3.2) we focused on other developments [52, 53, 59, 60] and nothing changed for the non-confluence part. For CoCo 2017 we have added Lemma 8.8 employing conditional narrowing to ConCon 1.5 as well as the certifiable CPF output for all of the non-confluence methods [61]. The TRS confluence checker CSI is also developed at the University of Innsbruck [44, 68]. The CTRS in Example 8.2 is from [48, Example 4.18] (320) and the one in Example 8.4 is from [6] (271).

The formalization of the methods described in this chapter can be found in the following IsaFoR theory files:

`thys/Conditional_Rewriting/Non_Confluence2.thy`

In the next chapter we will look at two supporting methods that facilitate the techniques described in this and the previous chapters.

Chapter 9

Supporting Methods

Sometimes directly using the methods for (non-)confluence that are described in earlier chapters is not possible. But there are sound methods that can "simplify" a given CTRS and make it amenable for them. In this chapter we will see two such methods.

The first one is about infeasibility. Already in Chapter 7 we have seen that infeasibility of CCPs can be expedient in analyzing confluence of CTRSs. Here we will learn that also infeasibility of conditions of rewrite rules can be beneficial for confluence analysis.

The second method can "reshape" certain conditional rewrite rules in such a way that other methods become more applicable.

9.1 Infeasible Rule Removal

If the conditions of a conditional rewrite rule are infeasible we can just remove this rule from the CTRS without changing the rewrite relation because the rule could never be fired anyway.

Definition 9.1 (Infeasible rule)**.** A conditional rewrite rule $\ell \to r \Leftarrow c$ is called *infeasible* if c is infeasible.

Why would anyone add an infeasible rule to a CTRS in the first place? We do not know the answer to this question but if we for example look at the Cops database we find that from 111 DCTRSs a stunning 11 systems contain infeasible rules. And that is just using fast and easy checks so there might be more where infeasibility of the conditions is not so obvious. All 11 DCTRSs with infeasible rules are from the literature. For instance the example below is due to Bergstra and Klop.

Example 9.2. Remember the DCTRS $\mathcal{R}$ from Example 8.4 consisting of the four rules

$$\mathsf{p}(\mathsf{q}(x)) \to \mathsf{p}(\mathsf{r}(x)) \qquad \mathsf{q}(\mathsf{h}(x)) \to \mathsf{r}(x) \qquad \mathsf{r}(x) \to \mathsf{r}(\mathsf{h}(x)) \Leftarrow \mathsf{s}(x) \approx 0 \qquad \mathsf{s}(x) \to 1$$

The right-hand side 0 of the single condition of the only conditional rule of $\mathcal{R}$ is an $\mathcal{R}$-normal form. The left-hand side $\mathsf{s}(x)$ does only rewrite to the different $\mathcal{R}$-normal form 1. Hence the condition is infeasible and we can just remove the rule, because it does not affect the rewrite relation of $\mathcal{R}$ in any way.

In principle we can use all the methods from Chapter 7 to remove infeasible rules before we employ any of the actual (non-)confluence checks. In the following example we utilize anc (see Section 7.4).

Example 9.3. Consider the CTRS $\mathcal{R}$ consisting of the two rules

$$\mathsf{h}(x) \to \mathsf{a} \qquad\qquad \mathsf{g}(x) \to \mathsf{a} \Leftarrow \mathsf{h}(x) \approx \mathsf{b}$$

The condition of the only conditional rewrite rule is infeasible because $\mathsf{h}(x)$ only rewrites to a and not to b. Unification fails to show that because $\mathsf{tcap}(\mathsf{h}(x)) = y \sim \mathsf{b} = \mathsf{tcap}(\mathsf{b})$. Fortunately we have $\mathsf{h} \not\trianglerighteq_{\mathsf{rs}} \mathsf{b}$ and hence the condition $\mathsf{h}(x) \approx \mathsf{b}$ is infeasible by Lemma 7.19. Or we can use exact tree automata completion. The underlying TRS $\mathcal{R}_u$ is linear and growing and hence we can construct the tree automata $\mathcal{A}_{\Sigma(\mathsf{h}(x))}$ and $\mathsf{anc}_{\mathcal{R}_u}(\mathcal{A}_{\Sigma(\mathsf{b})})$. The language of the intersection automaton is empty and the condition $\mathsf{h}(x) \approx \mathsf{b}$ is infeasible by Lemma 7.34.

9.2 Inlining of Conditions

In this section we look at another simple method that is inspired by inlining of let-constructs and where-expressions in compilers. We give a transformation on CTRSs which is often helpful in practice.

Definition 9.4 (Inlining of Conditions)**.** Given a conditional rewrite rule

$$\rho\colon \ell \to r \Leftarrow s_1 \approx t_1, \ldots, s_k \approx t_k$$

and an index $1 \leqslant i \leqslant k$ such that $t_i = x$ for some variable x, let $\mathsf{inl}_i(\rho)$ denote the rule resulting from *inlining* the i-th condition of ρ, that is,

$$\ell \to r\sigma \Leftarrow s_1\sigma \approx t_1, \ldots, s_{i-1}\sigma \approx t_{i-1}, s_{i+1}\sigma \approx t_{i+1}, \ldots, s_k\sigma \approx t_k$$

with $\sigma = \{x \mapsto s_i\}$.

Lemma 9.5. *Let $\rho \in \mathcal{R}$ and $s \approx x$ be the i-th condition of ρ. Whenever we have $x \notin \mathcal{V}(\ell, s, t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_k)$, then for $\mathcal{R}' = (\mathcal{R} \setminus \{\rho\}) \cup \{\mathsf{inl}_i(\rho)\}$ the relations $\to_{\mathcal{R}}^*$ and $\to_{\mathcal{R}'}^*$ coincide.*

Proof. We show $\to_{\mathcal{R},n} \subseteq \to_{\mathcal{R}',n}^*$ and $\to_{\mathcal{R}',n} \subseteq \to_{\mathcal{R},n}$ by induction on the level n. For $n = 0$ the result is immediate. Consider a step $s = C[\ell\sigma] \to_{\mathcal{R},n+1} C[r\sigma] = t$ employing rule ρ (for the other rules of $\mathcal{R}$ the result is trivial). Thus, $u\sigma \to_{\mathcal{R},n}^* v\sigma$ for all $u \approx v \in c$. In particular $s\sigma \to_{\mathcal{R},n}^* x\sigma$. Thus, using the IH, for each condition $u \approx v$ of $\mathsf{inl}_i(\rho)$ we have $u\sigma = s_j\{x \mapsto s\}\sigma \to_{\mathcal{R}',n}^* s_j\sigma \to_{\mathcal{R}',n}^* t_j\sigma = v\sigma$ for some $1 \leqslant j \leqslant k$. Hence, $\ell\sigma \to_{\mathcal{R}',n+1} r\{x \mapsto s\}\sigma \to_{\mathcal{R}',n+1}^* r\sigma$ and thus $s \to_{\mathcal{R}',n+1}^* t$.

Now, consider a step $s = C[\ell\sigma] \to_{\mathcal{R}',n+1} C[r\{x \mapsto s\}\sigma]$ employing rule $\mathsf{inl}_i(\rho)$. Together with the IH this implies that $u\sigma \to_{\mathcal{R},n}^* v\sigma$ for all conditions $u \approx v$ in $\mathsf{inl}_i(\rho)$. Let τ be a substitution such that $\tau(x) = s\sigma$ and $\tau(y) = \sigma(y)$ for all $y \neq x$. We have $s_i\tau = s\tau = x\tau = t_i\tau$ and $s_j\tau = s_j\{x \mapsto s\}\sigma \to_{\mathcal{R},n}^* t_j\sigma = t_j\tau$ for all $1 \leqslant j \leqslant k$ with $i \neq j$, since x occurs neither in s nor in the right-hand sides of conditions in $\mathsf{inl}_i(\rho)$. Therefore, $u \to_{\mathcal{R},n}^* v$ for all $u \approx v \in c$. In summary, we have $s = C[\ell\sigma] = C[\ell\tau] \to_{\mathcal{R},n+1} C[r\tau] = C[r\{x \mapsto s\}\sigma]$, concluding the proof. □

We are not aware of any mention of this simple method in the literature, but found that in practice, exhaustive application of inlining increases the applicability of other methods like infeasibility via tcap and non-confluence via plain rewriting: for the former inlining yields more term structure, which may prevent tcap from replacing a subterm by a fresh variable and thus makes non-unifiability more likely; while for the latter inlining may yield CCPs without conditions and thereby make them amenable to non-joinability techniques for plain term rewriting [68].

Example 9.6. Consider the quasi-decreasing ADCTRS $\mathcal{R}$ consisting of the following six rules:

$$\mathsf{min}(\mathsf{cons}(x, \mathsf{nil})) \to x \quad (1)$$
$$\mathsf{min}(\mathsf{cons}(x, xs)) \to x \Leftarrow \mathsf{min}(xs) \approx y,\ x < y \approx \mathsf{true} \quad (2)$$
$$\mathsf{min}(\mathsf{cons}(x, xs)) \to y \Leftarrow \mathsf{min}(xs) \approx y,\ x < y \approx \mathsf{false} \quad (3)$$
$$x < 0 \to \mathsf{false} \quad (4)$$
$$0 < \mathsf{s}(y) \to \mathsf{true} \quad (5)$$
$$\mathsf{s}(x) < \mathsf{s}(y) \to x < y \quad (6)$$

$\mathcal{R}$ has 6 CCPs, 3 modulo symmetry:

$$x \approx x \Leftarrow \mathsf{min}(\mathsf{nil}) \approx y,\ x < y \approx \mathsf{true} \quad (1,2)$$
$$x \approx y \Leftarrow \mathsf{min}(\mathsf{nil}) \approx y,\ x < y \approx \mathsf{false} \quad (1,3)$$
$$x \approx y \Leftarrow \mathsf{min}(xs) \approx z,\ x < z \approx \mathsf{true},\ \mathsf{min}(xs) \approx y,\ x < y \approx \mathsf{false} \quad (2,3)$$

To conclude confluence of the system it remains to check its CCPs. The first one, (1,2), is trivially context-joinable because the left- and right-hand sides coincide. Unfortunately, the methods used in ConCon are not able to handle either of the CCPs (1,3) and (2,3). So we are not able to conclude confluence of $\mathcal{R}$ just yet. But Rules (2) and (3) of $\mathcal{R}$ are both susceptible to inlining of conditions. For each of them, we may remove the first condition and replace y by $\mathsf{min}(xs)$ resulting in

$$\mathsf{min}(\mathsf{cons}(x, xs)) \to x \Leftarrow x < \mathsf{min}(xs) \approx \mathsf{true} \quad (2')$$
$$\mathsf{min}(\mathsf{cons}(x, xs)) \to \mathsf{min}(xs) \Leftarrow x < \mathsf{min}(xs) \approx \mathsf{false} \quad (3')$$

Now we actually arrive at the CTRS from Example 5.15 which is shown to be confluent in Chapter 5.

9.3 Certification and Implementation

Infeasible rule removal and inlining of conditions are both implemented in ConCon as a preprocessing step and are certifiable by CeTA.

For the certification of infeasible rule removal we can reuse machinery in ConCon and CeTA that is also used to show infeasibility of CCPs. The certificate just has to provide the infeasible rules together with the infeasibility proofs for their conditions. 90% of the time a system will not contain a single infeasible rule (see Chapter 10) and some of the available infeasibility checks are rather expensive (especially tree automata techniques). So in our implementation we only employ the *symbol transition graph* (see Section 7.2) method in this case because it is fast and lightweight.

To certify inlining CeTA requires ConCon to output the inlined CTRS $\mathcal{R}'$ together with a list of pairs that in the first component contain all modified rules and in the second component the corresponding list of conditions that have been inlined in this rule. Internally CeTA employs this information to reverse the inlining and finally checks if the result corresponds to the original input CTRS $\mathcal{R}$.

9.4 Chapter Notes

We have seen how to make CTRSs more amenable to the methods of the previous chapters by two simple techniques that on the one hand show infeasibility of rules and on the other hand simplify rules.

The CTRS in Example 9.3 is a smaller version of [53, Example 17] (495). The presentation in this chapter is based on sections of our publications [53, 54].

The formalization of the methods described in this chapter can be found in the following IsaFoR theory files:

```
thys/Conditional_Rewriting/Infeasibility.thy
thys/Conditional_Rewriting/Inline_Conditions.thy
thys/Conditional_Rewriting/Inline_Conditions_Impl.thy
```

In this and the previous chapters we have learned a lot about the underlying theory of the methods implemented in ConCon and how we formalized them in IsaFoR. In the following chapter we finally take a closer look at the CTRS confluence checker itself: we first give a high-level overview of the implementation and then learn about how to use it in practice.

Chapter 10

ConCon

This chapter describes the conditional confluence tool – ConCon – a fully automatic confluence checker for first-order conditional term rewrite systems.

The tool implements all of the quasi-decreasingness, (non-)confluence, and infeasibility criteria described in detail in the previous chapters. Additionally it is able to output certifiable proofs for its claims in a format that can be understood by the certifier CeTA. In the next sections we describe version 1.5 of ConCon.

10.1 General Design and Implementation

The tool ConCon 1.5 is written in Scala 2.12,[1] an object-functional programming language. Scala compiles to Java byte code and therefore is easily portable to different platforms. ConCon is available under the LGPL license and may be downloaded from:

`http://cl-informatik.uibk.ac.at/software/concon/`

The Scala sources are already prepacked in the ConCon distribution, so the only thing that has to be installed to use ConCon is a JDK 8 (or higher). The basic layout of ConCon is shown in Figure 10.1. When calling ConCon with an input file containing a CTRS it first tries to inline conditions (see Section 9.2) that resemble where-clauses or let-expressions of functional programs. Then the generalized tcap method employing the checks detailed in Sections 7.1, 7.2, and 7.3, is used to get rid of rules with infeasible conditions, because they don't add to the rewrite relation and hence may be safely discarded (see Section 9.1). Using the standard settings ConCon concurrently tries to apply one of five main methods to decide confluence:

A Try Theorem 5.14 from Chapter 5.

B Try Corollary 4.15 from Chapter 4.

C Try Theorem 3.4 from Chapter 3.

N Try the heuristics described in Chapter 8.

U Hand unconditional input over to CSI.

[1] `http://scala-lang.org`

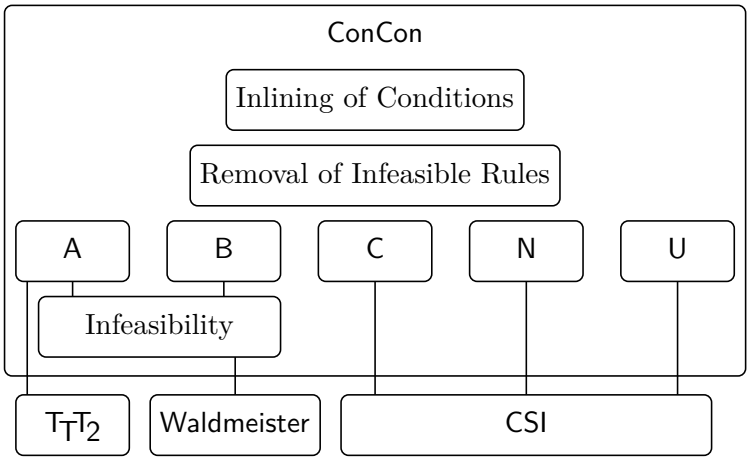


Figure 10.1: Schematic view of ConCon.

The first two methods (labeled A and B in the figure) may benefit from infeasible conditional critical pairs. So by default ConCon employs all implemented infeasibility criteria (see Chapter 7) to support them. Some of the methods make use of external tools. For example, to test quasi-decreasingness of a CTRS in method A the system is transformed to a TRS and then a call to an external termination checker for unconditional TRSs is issued. With the standard settings ConCon will try to call T$_T$T$_2$ but in fact any termination checker that reads the former plain text format of TPDB will do. See Section 10.4 on how to customize the external tools used by ConCon. Similarly, to check confluence of a CTRS via an unraveling in method C or non-confluence in method N by searching for non-joinable peaks, ConCon calls an external confluence checker for unconditional TRSs. Again, using the standard settings we use CSI but for the required confluence checks in method C any other confluence checker for unconditional TRSs that understands the right format maybe substituted. The non-joinability checks for method N are a different matter. They have been specifically designed for use by ConCon (see Section 8.2) and other confluence checkers would need to be modified to allow us to substitute them in CSI's place. Finally, one of the infeasibility methods uses the theorem prover Waldmeister.

10.2 Input and Output Formats

For input files ConCon supports two formats:

- The official, XML-based format[2] of TPDB used in the termination competition.
- The official plain text format[3] of Cops used in the confluence competition.

The latter is a modified version of the former plain text format[4] of TPDB. The modifica-

[2] `http://www.termination-portal.org/wiki/XTC_Format_Specification`
[3] `http://coco.nue.riec.tohoku.ac.jp/problems/ctrs.php`
[4] `http://www.lri.fr/~marche/tpdb/format.html`

```
(CONDITIONTYPE ORIENTED)
(VAR l x y)
(RULES
  le(0, s(x)) -> true
  le(x, 0) -> false
  le(s(x), s(y))  -> le(x, y)
  min(cons(x, nil)) -> x
  min(cons(x, l)) -> x | le(x, min(l)) == true
  min(cons(x, l)) -> min(l) | le(x, min(l)) == false
  min(cons(x, l)) -> min(l) | min(l) == x
)
```

Figure 10.2: System 292 from Cops in the plain text input format of ConCon.

tion concerns a new declaration `CONDITIONTYPE`, which may be set to `SEMI-EQUATIONAL`, `JOIN`, or `ORIENTED`. Although for now ConCon works on oriented CTRSs we designed the `CONDITIONTYPE` to anticipate future developments. In the conditional part of the rules we only allow `==` as relation, since the exact interpretation is inferred from the `CONDITIONTYPE` declaration. This modified plain text format is closer to the newer XML version and makes it very easy to interpret, say, a given join CTRS as an oriented CTRS (by just modifying the `CONDITIONTYPE`). As mentioned earlier, the modified plain text format is now the standard for the CTRS category of Cops. See Figure 10.2 for an example of an input file in the Cops format.

For output ConCon also supports two formats:

- a simple plain text format that is used by default and
- the CPF format[5] if the flag '`--cert`' (see Section 10.3 below) is set.

The plain text format should be self-explanatory. It sometimes contains references to papers and results therein. These can be resolved on the ConCon website.[6] The certification problem format on the other hand is an XML format used by the certifier CeTA. The CPF output serves two purposes. On the one hand it is easily parsable by CeTA and on the other hand we can generate a very nice and human readable HTML version from it. An example proof in the plain text format is depicted in Figure 10.3(a). In contrast the CPF output of the same proof is shown in Figure 10.3(b).

10.3 Usage

ConCon is operated through a command line interface that we present below. Just starting the tool without any options or input file as follows

```
java -jar concon-1.5.0.0.jar
```

[5] `http://cl-informatik.uibk.ac.at/software/cpf/`
[6] `http://cl-informatik.uibk.ac.at/software/concon/references.php`

```
Proof:
This system is confluent.
By \cite{SMI95}, Corollary 4.7 or 5.3.
This system is oriented.
This system is of type 3 or smaller.
This system is right-stable.
This system is properly oriented.
This is an overlay system.
This system is left-linear.
All 0 critical pairs are trivial or infeasible.
```

(a) Plain text format.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl"
  href="cpfHTML.xsl"?>
<certificationProblem>
 <input>
  <ctrsInput> ... </ctrsInput>
 </input>
 <cpfVersion>2.4</cpfVersion>
 <proof>
  <conditionalCrProof>
   <almostOrthogonal/>
  </conditionalCrProof>
 </proof>
 <origin>
  <proofOrigin> ... </proofOrigin>
 </origin>
</certificationProblem>
```

(b) CPF format.

Figure 10.3: Examples of confluence proofs for system 355 from Cops.

will output a short usage description. Note that for some input systems it might be necessary to increase the maximum stack frame for Java threads and/or the maximum heap size of Java. For example the call

```
java -jar -Xss20M -Xmx6G concon-1.5.0.0.jar
```

increases the maximum thread stack frame size to 20 megabyte and the maximum heap size to 6 gigabyte. In the following we will abbreviate this command by '`concon`'. Of course, the call

```
concon -h
```

also outputs the usage description. The flag '`--conf`' may be used to configure the employed confluence criteria. The flag takes a list of criteria which are tried in parallel. The available options are '`all`' to try all implemented methods, '`U`' to just check if the input system is unconditional and give it to an external confluence checker for TRSs, '`A`', '`B`', '`C`', for the respective methods of the same name described earlier, and finally '`N`' for method N. If one of the methods is successful ConCon immediately terminates returning its findings and the other checks are canceled.

Example 10.1. The call

```
concon --conf 'A B' 292.trs
```

prompts ConCon to use only methods A and B to check confluence of system `292.trs`.

In the same vein, the flag '`--inf`' may be used to configure which infeasibility criteria to use. The available options are '`no`' to not check infeasibility at all, '`tcap`' to use the

tcap check (see Section 7.1), '`gtcap`' to use a generalized tcap check with additional symbol transition graph and decomposition of reachability problems (see Sections 7.2 and 7.3), '`exeq`' to exploit equalities in the conditions, for example, to convert meets to joins (see Section 7.6), '`er`' to employ equational reasoning via the external tool Waldmeister (see Section 7.5), and finally '`etac`' to use exact tree automata completion (see Section 7.4). By default ConCon uses all the available confluence and infeasibility criteria as described in Section 10.1.

Example 10.2. When issuing the call

```
concon --inf 'tcap etac' 292.trs
```

ConCon will try to show confluence of system `292.trs` using all confluence methods but only using tcap and exact tree automata completion to check for infeasibility.

Additionally the inlining of conditions as well as the removal of infeasible rules maybe switched off by the flags '`--no-inlining`' and '`--no-infeasible-rule-removal`', respectively. One may always add a timeout at the end of ConCon's parameter list. The default timeout is 60 seconds.

Example 10.3. When calling ConCon with an input file `292.trs` like

```
concon 292.trs
```

it will just try to apply all available confluence and infeasibility criteria with the default timeout as explained above. The first line of the output will be one of '`YES`', '`NO`', or '`MAYBE`', followed by the input system, and finally a textual description of how ConCon did conclude the given answer.

One may use '`-a`', '`-s`', and '`-p`' to prevent output of the answer, the input system, and the textual description, respectively.

By setting the flag '`--cert`' ConCon is forced to only apply certifiable methods and to output a proof in the CPF format instead of just a textual description.

Example 10.4. Look at the following call:

```
concon --cert --conf A --inf tcap -a -s 292.trs
```

here we use ConCon to generate a certifiable proof in the CPF format that system `292.trs` is confluent by method A only using tcap for infeasibility. Because we directly want to pipe the output to CeTA we suppress the '`YES`' and also printing of the input system.

Critical Pairs. If one is only interested in the critical pairs of the system and which of them can be shown to be infeasible, one may use the following call

```
concon -c 292.trs
```

the '`-c`' flag causes ConCon to print all overlaps and the associated (conditional) critical pairs of the system, and indicates whether they could be shown infeasible. Because the infeasibility checks are done sequentially and may take some time ConCon might reach the timeout before printing all the CCPs. To just get a list of the CCPs without checking for infeasibility just use:

```
concon -c --inf no 292.trs
```

Quasi-Decreasingness. In order to check the input system for quasi-decreasingness the flag '`-q`' may be used. In addition one may use the option '`--ter`' together with one of the strings '`u`' or '`v`' to restrict the transformation to use for the termination check to U (see Definition 2.49) or V (see Definition 2.51), respectively. This method gives the transformed unconditional system to an external termination checker as determined in the settings file.

Transformations. The tool implements a considerable number of different transformations for CTRSs and sometimes it is helpful to just use ConCon as a translator. The flag '`-t`' may be used to tell ConCon to just apply a transformation and output the result. The flag takes a string parameter specifying which transformation to use. The rather extensive list of available options in alphabetical order:

`gn`	the structure-preserving transformation GN due to Gmeiner and Nishida.
`id`	the identity transformation.
`inline`	inlining of conditions (see Section 9.2).
`irrem`	infeasible rule removal (see Section 9.1).
`j`	the transformation J due to Antoy et al.
`ru`	the underlying TRS $\mathcal{R}_{\mathsf{u}}$ (see Definition 2.37).
`sr`	the structure-preserving SR-transformation due to Şerbănuţă and Roşu.
`u`	the unraveling U (see Definition 2.49).
`ucs`	the context-sensitive unraveling U_{CS} (see Definition 6.10).
`un`	the unraveling U_{n} due to Marchiori.
`uopt`	the optimized unraveling $\mathsf{U}_{\mathsf{opt}}$ due to Ohlebusch.
`v`	the transformation V (see Definition 2.51).
`xi`	the complexity-preserving transformation Ξ due to Kop et al.
`xi2`	a modified version of Ξ.
`xi2var`	a version of Ξ from where instead of anti-patterns we just use fresh variables.

This list of transformations may be produced by issuing the call

```
concon --transformations
```

Properties of CTRSs. Many syntactic criteria for CTRSs, like proper-orientedness or weak left-linearity, are tedious to check by hand. Other properties of interest, like quasi-decreasingness, are undecidable. Executing the call

```
concon -l 292.trs
```

results in a list of selected properties of the input CTRS. But this is just a quick and easy way to show some properties. The command

```
concon --properties
```

yields a list of all properties that ConCon is able to check. Now, using the flag '`--prop`' and a combination of these properties by the usual logical connectives '`&`' (and), '`|`' (or), and '`!`' (not) one can construct more involved queries.

Example 10.5. The query

```
concon --prop 'oriented & !(orthogonal | right_stable)' 292.trs
```

checks whether system `292.trs` is indeed an oriented CTRS which is neither orthogonal nor right-stable.

10.4 Settings

As explained earlier, some of ConCon's methods need external programs to work properly. All in all ConCon utilizes three different external programs. To make these programs available to ConCon the paths to them as well as their parameters can be adjusted in the file `concon.ini`, which should reside in the same directory as the `concon` executable. The contents of `concon.ini` default to:

```
confluence_checker=csi -s 'AUTO[30]' -ext trs -
termination_checker=ttt2 -s 'COMP[30]' -ext trs -
non_joinability_checker=csi
  -s '(nonconfluence -nonjoinability -steps 0 -tree)[30]'
  -C RT -ext trs -
confluence_certifier=csi -ext trs -cpf -s 'CERT_ALL[30]' -
termination_certifier=ttt2 -ext trs -cpf -s 'COMPCERT[30]' -
non_joinability_certifier=csi -ext trs -cpf
  -s '(nonconfluence -nonjoinability -steps 0 -tree -cert)[30]'
  -C RT -
waldmeister=waldmeister
```

Let us look at the entries in more detail:

- The entry `confluence_checker` has to point to a TRS confluence checker that is able to read a TRS in the plain text format of TPDB from standard input. The first line of its output is expected to be '`YES`' (if the given TRS is confluent), '`NO`' (if the given TRS is non-confluent), or '`MAYBE`'. This tool is used in methods C and U and defaults to the TRS confluence checker CSI.
- The `termination_checker` works in a similar matter only that the property to check is termination of course. This checker is used to show quasi-decreasingness (used in method A or on its own). It defaults to $\mathsf{T_TT_2}$.

- The `non_joinability_checker` works a little bit differently. Here the expected input has to also contain the two terms that we want to check for non-joinability (explained in more detail in Section 8.2). This checker is used in method N and also defaults to CSI.
- The corresponding entries for the certifiers, that is, `confluence_certifier`, `termination_certifier`, and `non_joinability_certifier`, work similarly to their checker counterparts only that in addition to the answer ('`YES`', '`NO`', or '`MAYBE`') they also have to produce a proof of their claim in the CPF format. The certifiers are used in place of the checkers if ConCon's '`--cert`' flag is set.
- Finally, `waldmeister` is used in one of ConCon's infeasibility methods (see Section 7.5). Here the communication protocol is tailored to the automatic theorem prover Waldmeister.

The same default values for these settings are also stored inside ConCon and are used if no `concon.ini` file is provided.

10.5 Web Interface

In addition to the command line version there is also an easy to use web interface available on the ConCon website

`http://cl-informatik.uibk.ac.at/software/concon/webint.php`

This web interface is depicted in Figure 10.4. You can use it in three simple steps: first under "1. Input Conditional Term Rewrite System" select a CTRS from the "select example" pull-down menu or upload one via the "Browse..." button. Next under "2. Select Action" choose which action you want ConCon to perform. By default this will be to check confluence of the input CTRS. Other options are to list its critical pairs, to list some other interesting properties of it, to check quasi-decreasingness, or to output a transformed system using some of the transformations implemented in ConCon. When choosing the default action there are some further settings to choose. These concern the non-confluence methods, inlining, infeasible rule removal, the infeasibility methods, and certification. For more details on that see Sections 10.1 and 10.3. Finally, under "3. Start ConCon" press the "execute" button to start ConCon. The timeout is fixed to 60 seconds.

10.6 Troubleshooting

If at any point something goes wrong with ConCon, like strange Java exceptions or the tool does not do what you would expect, don't hesitate to contact me at

`concon@informatik.uibk.ac.at`

It might also be useful to switch on the logging facilities of ConCon by setting one of the flags '`--error`', '`--debug`', or '`--trace`' while exploring misbehavior and then send along the output (and of course the input file).

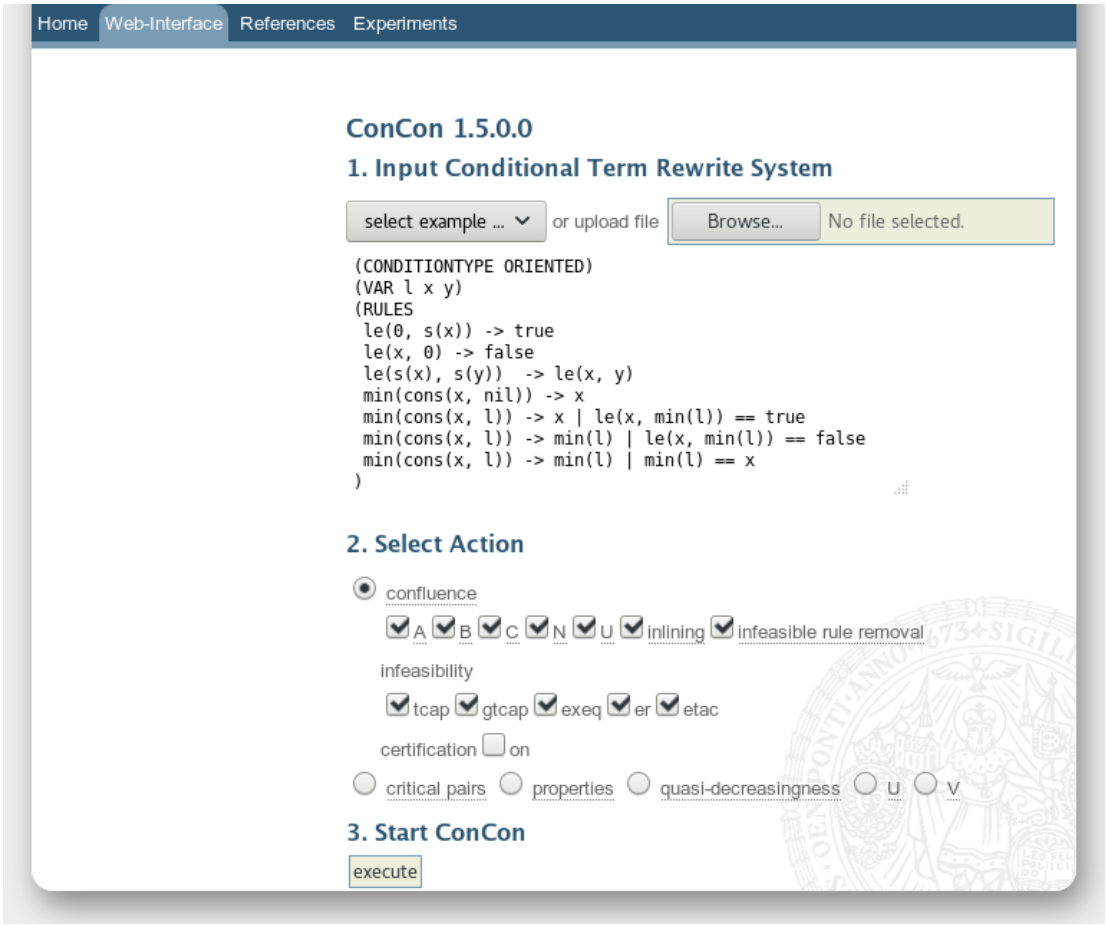


Figure 10.4: ConCon's web interface.

10.7 Additional Tool Infrastructure

Over the years ConCon has grown tremendously and some of the features are mainly used to manage CTRS files and to arrange them for experiments and are no longer directly connected to proving confluence of CTRSs. For that reason we implemented several entry points to the ConCon codebase which provide additional functionality for particular tasks.

The default main class `concon.ConCon` provides the tool described above. By just switching the main class with the command:

```
jar ufe concon-1.5.0.0.jar concon.CTRSProperties
```

ConCon is "transformed" in a tool that can check various properties of several CTRS files at once. The properties are specified as explained earlier but in contrast to ConCon's '`--prop`' flag this tool now works on a provided directory containing CTRS files and checks, filters, or just counts these files according to their properties.

Another useful application is the "CTRS transformer" that can be generated by the command:

```
jar ufe concon-1.5.0.0.jar concon.CTRSTransformer
```

This tool allows to apply any of ConCon's available transformations to all CTRS files in a given directory.

Finally, the "CTRS similarity checker" takes a directory containing CTRS files and groups them by similarity. Two systems are considered to be similar if they have the same number of variables, the same number of function symbols with the same arity, and the same number of conditional and unconditional rules. While this is just an ad hoc method it is for example useful to find duplicates which just use different function symbols in a collection of CTRSs. The similarity checker can be produced by typing:

```
jar ufe concon-1.5.0.0.jar concon.CTRSSimilarity
```

10.8 Chapter Notes

In this chapter we presented version 1.5 of our CTRS confluence checker ConCon, for an earlier version see [58]. We have seen the general layout of this tool, some example use cases, and how to set up the paths and parameters for the external programs used by ConCon. For more details on the certifier CeTA and the CPF format used by it see [65] and [56]. The transformations U, V, and U_{CS} are described in this thesis (see Definition 2.49, Definition 2.51, and Definition 6.10, respectively). To read more about the other transformations that are supported by ConCon see the following references: the unraveling U_{n} can be found in [49, page 190, Definition 7.2.11], the optimized unraveling $\mathsf{U}_{\mathsf{opt}}$ can be found in [49, page 213], the structure-preserving transformation GN can be found in [26] and is based on the SR-transformation from [11], transformation J can be found in [2, page 25, Definition 13], and finally, different versions of the complexity-preserving transformation Ξ can be found in [32, 33].

Work on ConCon started in 2013 when I started my PhD studies. I was just in the process of learning about conditional and constrained rewriting. That my work should be about confluence was already clear from the outset. After a few weeks of initial reading I soon decided to concentrate on confluence of conditional rewriting and left constrained rewriting for someone else to pursue. Because I had previously – during my Bachelor and Master studies – built an automatic completion tool for unconditional term rewriting (named KBCV, see [57, 62, 63]) in the programming language Scala, I also wanted to use the same language for my new endeavor. On the one hand, the functional features of Scala are very handy when doing symbolic computation like in term rewriting. On the other hand, Scala compiles to Java byte code and runs on the Java virtual machine and hence may be used on many platforms. I wanted to have a short and easy to remember name for my tool. So I went for the obvious ***Con****fluence of* ***Con****ditional Term Rewrite Systems* – ConCon.

For the next year or so I went through the literature and tried to implement some of the confluence criteria I found. An initial working version of ConCon was soon up

and running. Luckily, around that time also other researchers became interested in automatic tools for confluence of CTRSs and so over the next three successful years ConCon had to face several competitors. Already at ConCon's first CoCo participation in 2014 the tool CO3 by Naoki Nishida et al. from Nagoya University tested its mettle against my tool. The following year CoScart by Karl Gmeiner from UAS Technikum Wien joined the fray. No need to mention that without them developing ConCon would only have been half the fun.

Early experiments already showed that handling of infeasible conditional critical pairs would be essential for the applicability of the methods used by ConCon. So a lot of time and effort went into researching, implementing, and testing of infeasibility criteria. Not all of these made it in the current version of my tool. For example, I implemented tree automata completion à la Timbuk (see [5]) and even conditional tree automata completion (see [16]) but in tests these methods could not outperform easier but faster methods like those presented in Chapter 7.

In the second half of my PhD research I mainly focused on formalizing the methods used by ConCon in order to be able to certify its output. To date ConCon is still the only conditional confluence checker with certifiable output.

In the next chapter we present the results of our extensive experiments comparing the different confluence, infeasibility, and quasi-decreasingness methods on the confluence problems database.

Chapter 11

Experimental Results

In the previous chapters we have first established the theory underlying several confluence, infeasibility, and quasi-decreasingness methods and subsequently illustrated how we implemented them in our CTRS confluence checker ConCon. Now we want to compare these methods experimentally in order to get empirical evidence on their strengths and weaknesses.

Problem Collection. When I started to look for some input to test my newly created tool ConCon in 2013 I soon noticed that there were no reasonable large collections of CTRSs available. At that time the termination problems data base – TPDB[1] (version 8.0.7) – for example featured a stunning total of 7 CTRSs. I already had been aware of the annual confluence competition – CoCo[2] – for some time. Since some of CoCo's initiators are working in the same research group as me it was easy to convince them to add a conditional category for CoCo 2014. So I went through the literature again and collected as many CTRSs as I could find and submitted them to the newly established conditional category of Cops[3] (the confluence problems database). Over time many other people have also contributed to this category and at the time of writing Cops contains 158 CTRSs. Well, not all of the CTRSs in Cops are *oriented*, there are 21 *join* CTRSs and 8 *semi-equational* CTRSs. As mentioned earlier ConCon works on *oriented* CTRSs, so in the sequel we will concentrate on Cops' 129 *oriented* CTRSs. For some of our experiments we have to further restrict the kind of CTRSs we are looking at but we will mention these additional restrictions when we impose them.

Test Environment. All experiments have been carried out on a 64bit GNU/Linux machine with 12 Intel® Core™ i7-5930K processors clocked at 3.50GHz and 32GB of RAM. The kernel version is `3.16.0-4-amd64`. The version of Java on this machine is 1.8.0_131. We had to increase the stack size used by ConCon to 20MB using the JVM flag '`-Xss20M`' to prevent stack overflows caused by parsing deep terms like in Cops system `313`.

Tools. Here is a list (in alphabetical order) of all tools that we have used in our experiments, their versions (if known), and where to get them.

[1] `http://termination-portal.org/wiki/TPDB`
[2] `http://coco.nue.riec.tohoku.ac.jp/`
[3] `http://cops.uibk.ac.at/?q=ctrs`

AProVE	`http://aprove.informatik.rwth-aachen.de`
CeTA 2.30	`http://cl-informatik.uibk.ac.at/software/ceta`
CO3 1.3	`http://www.trs.cm.is.nagoya-u.ac.jp/co3`
ConCon 1.5	`http://cl-informatik.uibk.ac.at/software/concon`
CoScart	`https://github.com/searles/RewriteTool`
CSI 1.0	`http://cl-informatik.uibk.ac.at/software/csi`
MU-TERM 5.13	`http://zenon.dsic.upv.es/muterm`
NaTT 1.6	`http://www.trs.cm.is.nagoya-u.ac.jp/NaTT`
T_TT_2 1.16.2	`http://cl-informatik.uibk.ac.at/software/ttt2`
VMTL 1.3	`https://www.logic.at/vmtl`
Waldmeister July 99	`http://www.waldmeister.org`

Overview. We will proceed in a top-down fashion and first present results on ConCon's overall power compared to the other tools in CoCo's conditional track. Next we will look into each (non-)confluence method implemented in ConCon in more detail and compare them to each other. In the course of these investigations we will also look more closely at ConCon's quasi-decreasingness check and compare it to other dedicated termination tools. Finally, an in-depth examination of the various infeasibility methods will top off our experiments.

11.1 Comparing Confluence Tools for CTRSs

There are currently three automatic confluence tools for CTRSs that take part in the conditional track of the annual confluence competition. Besides ConCon these are CO3 and CoScart. In this section we want to compare the strengths and weaknesses of these three tools. To this end we compare the versions of CO3 and CoScart from CoCo 2016 to ConCon 1.5 and run them on the 129 oriented CTRSs of Cops with a timeout of 60 seconds.

The ***Co****nverter for proving* ***Co****nfluence of* ***Co****nditional TRSs* – CO3 – is being developed since 2014 by Nishida et al. at Nagoya University. It is written in OCaml and it uses a transformational approach employing the unraveling U as well as the structure-preserving transformation SR to prove confluence of CTRSs. In this sense it is mainly a converter that only uses simple and lightweight functions to verify confluence and termination of TRSs. Because of that it is very fast. CO3 also has some non-confluence checks. Unfortunately it does not provide a detailed proof output but just the minimal verdict ('`YES`', '`NO`', or '`MAYBE`').

CoScart on the other hand is being developed since 2015 by Gmeiner at Technikum Wien. Like ConCon it is written in Scala. In contrast to ConCon it is a stand-alone-tool and does not rely on any other software. For example CoScart comes with its own internal automatic termination prover employing the dependency pair method in combination with argument filtering. To show confluence of CTRSs it uses the structure-preserving transformation GN, modularity of confluence, a Knuth-Bendix criterion, and development-closed critical pairs of left-linear TRSs. CoScart cannot show

non-confluence.

Both CO3 and CoScart are not able to produce certifiable CPF output and hence only ConCon's output is certifiable by CeTA. Table 11.1(a) summarizes the results of this set of experiments.

The first line labeled 'confluent' shows how many 'YES'-instances have been found by each tool. There are three things to notice. First of all CO3 reportedly (by the tool's author) outputs some false positives (systems 279, 351, 404, 410, 411, and 489). These six systems are already subtracted from CO3's confluence results and separately listed in the line labeled 'erroneous'. For four of these systems (351, 404, 410, 411) we can confirm that the 'YES'-answer by CO3 is indeed wrong, because we get certified non-confluence for them using ConCon. For the other two ConCon does not find a proof. For system 522 CO3 is the only tool that claims confluence. Although we cannot confirm this because ConCon does not find a proof here either, it could be that this is also a false positive. Secondly the 'YES'-instances of CoScart are completely subsumed by ConCon. The relative strength of the three tools in proving confluence with respect to each other is shown in the figure depicted in Table 11.1(b). Finally, we see that there is a difference of seven systems between ConCon also using uncertifiable methods and ConCon plus CeTA only counting certified 'YES'-instances. Here system 286 is shown to be confluent by Theorem 3.4 which is not certified and the remaining six systems (340, 361, 406, 407, 409, and 440) all rely on Waldmeister to show infeasibility of some of their conditional critical pairs (see Section 7.5).

Now for non-confluence (second line) we see that here all 42 'NO'-instances of ConCon are certifiable by CeTA. While CoScart does not support non-confluence at all CO3 is able to show 27 systems non-confluent. There is one system (293) where ConCon cannot conclude non-confluence but CO3 can. It seems that CO3 employs methods related to (but more powerful than) our symbol transition graph and hence in this case is able to show infeasibility of a rule where ConCon fails.

As shown in the fourth line of Table 11.1(a) CO3 and CoScart cannot handle 4-CTRSs whereas ConCon can show (and certify) non-confluence of all four 4-CTRSs (309, 314, 318, and 320) in Cops. On the first three of these 4-CTRSs all of ConCon's non-confluence methods succeed whereas on system 320 only Lemma 8.1 is successful.

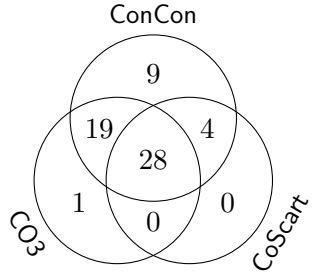
11.2 Comparing ConCon's Confluence Methods

In this second part of our experiments we will take a closer look at ConCon's confluence methods in comparison to each other. We use the same 129 oriented CTRSs as in the previous section and again set the timeout to one minute. Figure 11.1 breaks down the three confluence methods Theorem 5.14 (A), Corollary 4.15 (B), and Theorem 3.4 (C) as a function of other settings of the tool.

If we look at ConCon's absolute power, employing the supporting methods from Chapter 9, infeasibility of CCPs from Chapter 7, and also uncertified methods (compare Sections 7.5 and 3.1) we get the numbers in the Venn diagram depicted in Figure 11.1(a). There is a total of 60 systems that are shown to be confluent. We see that on its own

	CO3	CoScart	ConCon	ConCon + CeTA
confluent	48	32	60	53
non-confluent	27	0	42	42
erroneous	6	0	0	0
unsupported	4	4	0	0
open	44	93	27	34

(a) Confluence results per tool (absolute).



(b) 'YES'-instances (relative).

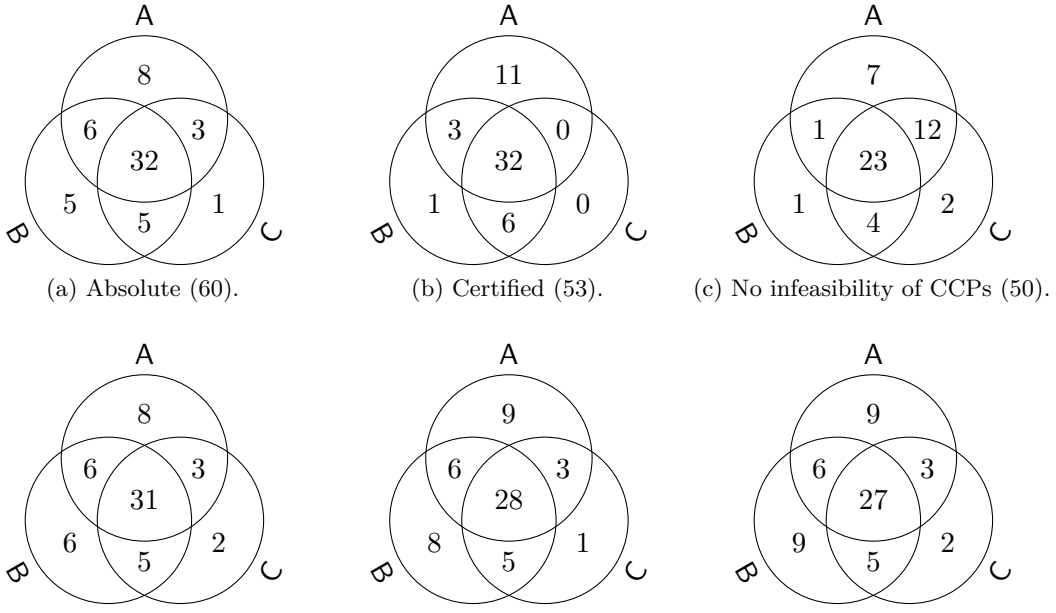
Table 11.1: Confluence results on all 129 oriented CTRSs from Cops.

method A is the strongest, succeeding on 49 CTRSs, closely followed by method B, which succeeds on 48 CTRSs, and finally method C, which can show confluence of 41 systems. Moreover, there are 8 CTRSs where only method A succeeds, 5 where only method B succeeds, and 1 where only method C succeeds.

Now if we restrict to only use certifiable methods, that is, no Waldmeister to show infeasibility of conditional critical pairs and for method C Theorem 3.1 instead of Theorem 3.4, the results can be found in Figure 11.1(b). First notice that the total is reduced to 53 confluent systems. Compared to Figure 11.1(a) methods A and C lose 3 systems each whereas method B loses 6. Concerning method C systems 286, 316, and 319 are all weakly-left linear (and Theorem 3.4 succeeds if we use uncertified methods) but their unravelings are not left-linear and hence Theorem 3.1 is not applicable. Of these three systems 286 is the one where previously only method C succeeded. The losses for methods A and B are all due to the exclusion of Waldmeister (see Section 7.5) from showing infeasibility of conditional critical pairs.

Next, in Figure 11.1(c), we see the confluence results if we switch off all infeasibility checks for conditional critical pairs. The total of confluent systems is reduced to 50. As expected, because it does not profit from infeasible CCPs, method C is unaffected by this measure. Method A loses 6 systems, it does rely on infeasibility of CCPs but in addition it also employs context-joinability and unfeasibility of CCPs (see Definition 5.12). On the other hand, method B is affected the most, losing 19 systems, because without infeasibility of CCPs method B is reduced to a purely syntactic check that relies on the absence of any non-trivial critical pairs.

Switching off inlining (but still using infeasible rule removal) surprisingly increases the overall number of confluent systems to 61 (see Figure 11.1(d)). Inlining may give some terms more structure, which is good to show non-reachability using tcap but if we need tree automata completion to show infeasibility of a conditional critical pair more structure may lead to a larger tree automaton that we have to compute and hence to a timeout. Apparently this is what happens for system 529 and while ConCon times out when inlining is used it succeeds when we switch it off. We also want to remark that although without inlining ConCon is able to produce a certificate for system 529, CeTA is



(a) Absolute (60). (b) Certified (53). (c) No infeasibility of CCPs (50).

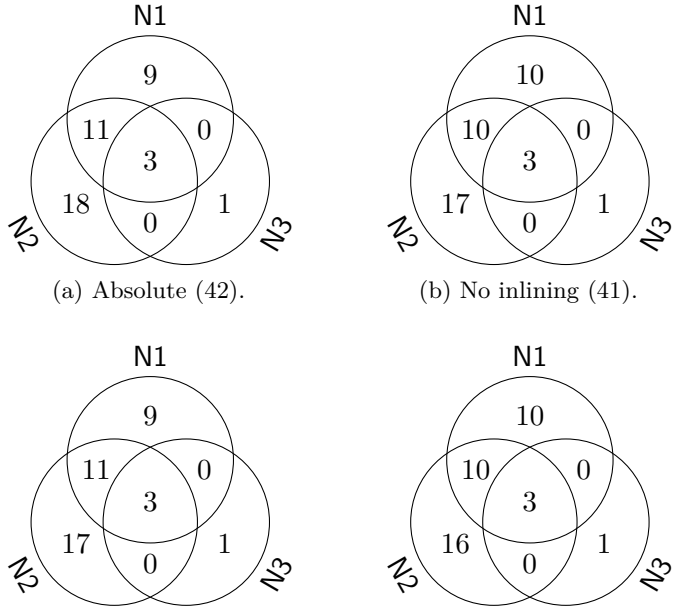
(d) No inlining (61). (e) No infeasible rule removal (60). (f) No supporting methods (61).

Figure 11.1: Comparison of ConCon's confluence methods.

not able to verify this because to do so it would have to compute 21^{11} state substitutions to check the given tree automaton and this causes a stack overflow. Besides, system 493 is now only proven confluent by method C because without inlining ConCon's infeasibility checks do not succeed on two of its CCPs.

Now, if we switch off infeasible rule removal (but keep inlining) the overall number of confluent systems is not affected (see Figure 11.1(e)) but system 264 which consists of a single infeasible rule is empty when employing infeasible rule removal (and hence handled by all confluence methods) but the rule is not left-linear and not even weakly left-linear and hence cannot be handled by methods B and C. System 287 consists of a single infeasible rule too. This rule is not weakly left-linear and ConCon is neither able to show it quasi-decreasing nor strongly deterministic but method B still works. Now 336 consists of three rules of which two are infeasible. Without infeasible rule removal the system is not deterministic but extended properly oriented and so only method B works. Finally, system 495 has four rules of which one is infeasible. When removing it methods B and C work but without infeasible rule removal only method B works because the system's unraveling is shown is non-confluent.

For sake of completeness we also include Figure 11.1(f) which shows the results when switching off both inlining and infeasible rule removal. As expected the results are basically a combination of the previous two diagrams.



(a) Absolute (42). (b) No inlining (41).

(c) No infeasible rule removal (41). (d) No supporting methods (40).

Figure 11.2: Comparison of ConCon's non-confluence methods.

11.3 Comparing ConCon's Non-Confluence Methods

In this section we want to take a closer look at ConCon's non-confluence methods in comparison to each other. We use the same 129 oriented CTRSs as in the previous sections and the same timeout of one minute. Results for the three non-confluence methods Lemma 8.8 (N1), Lemma 8.3 (N2), and Lemma 8.1 (N3) of ConCon are summarized in Figure 11.2. Since all of these methods are certifiable we do not need a separate diagram for that case.

When it comes to non-confluence ConCon is able to handle 42 systems when employing the supporting methods from Chapter 9. Figure 11.2(a) compares the three non-confluence methods of ConCon. On its own method N2 is the strongest succeeding on 32 systems, followed by method N1 succeeding on 23 systems, and finally method N3 which is specifically targeted at 4-CTRSs and hence can only handle the 4 systems of this kind.

As shown in Figure 11.2(b) when we switch off inlining method N2 loses 2 systems. Inlining transforms system 351 to an unconditional system and cannot be handled by ConCon without this method. System 353 can only be handled by method N1 without inlining. The total number of non-confluent systems is reduced to 41.

Similarly, as shown in Figure 11.2(c) when switching off infeasible rule removal method N2 loses one system. This system, 271, is reduced to a TRS when employing infeasible rule removal and cannot be handled by ConCon without this method.

Finally, the results for switching off both supporting methods are depicted in Fig-

	conditional $\mathcal{R}$				$\mathsf{U}_{\mathsf{CS}}(\mathcal{R})$		
	AProVE	ConCon	MU-TERM	VMTL	AProVE	MU-TERM	VMTL
quasi-decreasing	84	83	82	84	82	82	82
non-quasi-decreasing	–	–	14	–	–	–	–

	$\mathsf{U}(\mathcal{R})$					
	AProVE	MU-TERM	NaTT	$\mathsf{T_TT_2}$	VMTL	total
quasi-decreasing	85	82	81	82	82	88
non-quasi-decreasing	–	–	–	–	–	14

Table 11.2: Quasi-decreasingness results on all 111 DCTRSs from Cops.

ure 11.2(d). The total number of non-confluent systems is reduced to 40 because as explained earlier we lose systems 351 and 271. Unlike in the case of the confluence methods, where inlining causes ConCon to lose one system, in the non-confluence case it is exclusively beneficial.

11.4 Comparing Termination Tools for CTRSs

In this section we want to present results on quasi-decreasingness. To this end we conducted experiments on the 111 DCTRSs contained in Cops using the methods of ConCon and various automated termination tools. Of these, AProVE, MU-TERM, and VMTL are able to directly show quasi-decreasingness and MU-TERM is the only tool that can show non-quasi-decreasingness. AProVE, MU-TERM, and VMTL can also handle CSRSs and we used them in combination with $\mathsf{U}_{\mathsf{CS}}(\mathcal{R})$. Finally, we also ran the previous tools together with NaTT and $\mathsf{T_TT_2}$ on $\mathsf{U}(\mathcal{R})$. The results for a timeout of one minute are shown in Table 11.2.

There are several points of notice. AProVE together with $\mathsf{U}(\mathcal{R})$ yields the most quasi-decreasing systems (85). Interestingly, AProVE cannot show quasi-decreasingness of system 362 directly, although it succeeds (like all other tools besides NaTT) if provided with its unraveling. Moreover, systems 266, 278, and 279 can be shown to be quasi-decreasing by AProVE if we use $\mathsf{U}(\mathcal{R})$ but not if we use $\mathsf{U}_{\mathsf{CS}}(\mathcal{R})$ (even if we increase the timeout to 5 minutes). On system 363 only MU-TERM succeeds (in the direct approach). If we compare MU-TERM on conditional systems to MU-TERM with $\mathsf{U}_{\mathsf{CS}}(\mathcal{R})$, the direct method succeeds on system 360 but not on system 329. Conversely, when using $\mathsf{U}_{\mathsf{CS}}(\mathcal{R})$ it succeeds on system 329 but not on system 360. Moreover, MU-TERM seems to have some problems with systems 278 and 342, generating errors in the direct approach. With $\mathsf{U}_{\mathsf{CS}}(\mathcal{R})$ AProVE, MU-TERM, and VMTL succeed on the exact same 82 systems. On system 357 only VMTL, using the direct approach, succeeds. Employing $\mathsf{U}(\mathcal{R})$, NaTT succeeds on 81 systems, this is subsumed by $\mathsf{T_TT_2}$, succeeding on 82 systems, which in turn is subsumed by AProVE, succeeding, as mentioned above, on 85 systems. ConCon

	I1	I2	I3	I4	I5	total
infeasible	11	14	1	12	23	30
timeout	0	0	0	27	44	45
open	122	119	132	94	66	58

Table 11.3: Infeasibility results on 133 CCPs originating from Cops.

which is basically TTT2 on U($\mathcal{R}$) plus inlining and infeasible rule removal handles one system more (386) than TTT2 due to the removal of an infeasible rule. In total 88 systems are shown to be quasi-decreasing, 14 systems to be non-quasi-decreasing, and only 9 remain open. These are systems 288, 311, 312, 330, 337, 342, 405, 499, and 529.

In our context quasi-decreasingness is mainly interesting for applying method A. So the question is whether we can improve upon the current 49 CTRSs (see Figure 11.1(a)) that method A is able to show confluent if we would employ other tools for quasi-decreasingness. We will concentrate on the 27 CTRSs that cannot be handled by ConCon so far (see Table 11.1) and ignore those which are handled by one of the other methods. Of these 27 CTRSs 15 are also ADCTRSs and hence in principle amenable to method A. In turn 7 of these ADCTRSs are shown to be quasi-decreasing but only one of these, system 279, cannot be already handled by the quasi-decreasingness methods of ConCon. This system has 21 CCPs (modulo symmetry). Most of them cannot be handled by the context-joinability, unfeasibility, or infeasibility methods of ConCon so knowing that it is quasi-decreasing unfortunately does not help. This answers our question in the negative.

11.5 Comparing ConCon's Infeasibility Methods

In this last section of our experiments we want to compare the five infeasibility methods of ConCon, that is, the tcap check (I1, see Section 7.1), the generalized tcap check with symbol transition graph analysis and decomposition of reachability problems (I2, see Sections 7.2 and 7.3), exploiting equalities in the conditions (I3, see Section 7.6), equational reasoning using Waldmeister (I4, see Section 7.5), and finally exact tree automata completion (I5, see Section 7.4). To this end we take all CCPs of the 129 oriented CTRSs from Cops. We only keep the ones which have non-empty conditions and we count modulo symmetry. We are left with 133 CCPs for which we try ConCon's infeasibility methods separately with a timeout of 60 seconds. The results of this experiments are listed in Table 11.3.

The first line, labeled 'infeasible', lists the number of CCPs which could be shown to be infeasible with each method. The row labeled 'open' gives the number of CCPs for which infeasibility could not be shown within the time limit. On its own method I5 is the strongest showing infeasibility of 23 CCPs but it also is very computation intensive and causes the most timeouts (44). There are 10 CCPs where only method I5 is able to show infeasibility. Next, method I2 shows 14 CCPs infeasible and has no timeouts.

Method I4 can show 12 CCPs infeasible and causes 27 timeouts. There are 7 CCPs where only method I4 is able to show infeasibility. Method I1 shows 11 CCPs infeasible and has no timeouts. Finally, method I3 can only show infeasibility of a single CCP and also has no timeouts. In summary, all methods together succeed on 30 CCPs and time out on 45 CCPs.

11.6 Chapter Notes

We have seen several experiments on the one hand comparing ConCon to other confluence tools for CTRSs and dedicated termination tools and on the other hand on the different aspects of ConCon itself.

For more details on the other tools used in this chapter follow their respective references: AProVE [21], CeTA [65], CO3 [47], CoScart [24], CSI [68], MU-TERM [1], NaTT [67], T_TT_2 [34], VMTL [50], and Waldmeister [17]. As mentioned earlier MU-TERM is the only tool that can show non-quasi-decreasingness [39].

This brings us to the final chapter of this thesis where we will reflect on the findings of all previous chapters and lay out some possible future work.

Chapter 12

Conclusion

We set out to provide a fully automatic tool to reliably check confluence of CTRSs. In the last few chapters we presented an overview of the results that we have formalized in IsaFoR. Through code generation these are now certifiable by the certifier CeTA. We further provided a system description of ConCon 1.5 which implements all of the described methods and is able to produce output that is readable (and certifiable) by CeTA for most of them.

12.1 Summary

We started this journey by giving an introduction to conditional term rewriting, related topics, and interactive theorem proving in Chapter 2. Then we presented the three main confluence methods used by ConCon: the result that a DCTRS is confluent if its unraveling is left-linear and confluent in Chapter 3, that almost orthogonal (modulo infeasibility), right-stable, and extended properly oriented CTRSs are confluent in Chapter 4, and finally that quasi-decreasing, strongly deterministic CTRSs are confluent if all their conditional critical pairs are joinable in Chapter 5. In Chapter 6 we presented some methods to show quasi-decreasingness of DCTRSs, which we mainly require to make the method of the preceding chapter applicable. The topic of Chapter 7 were several methods to check for non-reachability between terms. In our context these are used to get infeasibility of conditional critical pairs as well as conditional rules in order to make the methods of Chapters 4 and 5, that both rely on critical pair analysis, more applicable. In detail the non-reachability checks are by unification, the symbol transition graph employing decomposition of reachability problems, exact tree automata completion, equational reasoning, and the exploitation of certain equalities in the conditions. Chapter 8 presented some checks, most notably a method employing conditional narrowing, to find witnesses for non-confluence of CTRSs. After that we touched on two supporting methods in Chapter 9. One that employs some of the results of Chapter 7 to get rid of infeasible rules and another that inlines certain conditions of rules. The whole of Chapter 10 was dedicated to ConCon. It started with an overview of the implementation, then provided some details on the supported input and output formats, an extended section on how to use the tool in practice, some notes on ConCon's settings, the web interface, and some additional information. Finally, Chapter 11 presented our experiments on the confluence problems database. We first compared ConCon to the other tools of CoCo's conditional category. Next we compared ConCon's three confluence

methods to each other. Following that, we presented experimental results on ConCon's non-confluence methods. The next section was dedicated to a comparison of different tools for quasi-decreasingness, before we investigated ConCon's infeasibility methods in some detail.

12.2 Formalization and Implementation

Besides compiling this thesis most of the work went into the formalization of the presented results. Here is a rough impression of the involved effort: our formalization comprises 92 definitions, 37 recursive functions, and 518 lemmas with proofs, on approximately 10,000 lines of Isabelle code (in addition to everything that we could reuse from the IsaFoR library).

To give some additional measure on how much work went into the formalization we take a look at the *de Bruijn factor* of some of its parts. The de Bruijn factor has been defined by Freek Wiedijk[1] to be the quotient of the size of a formalization of a mathematical text and the size of its informal original. Because our formalization depends on a lot of prior results from IsaFoR the scope of a specific result is not so easy to define and hence somewhat arbitrary. Nevertheless, when comparing the textual description in this thesis to our formalization in IsaFoR (without the additional setup for code generation and CeTA's parser) we get the following (approximate) numbers: the results of Section 7.4 have a de Bruijn factor of 9.8, Theorem 4.14 has a de Bruijn factor of 4.2, and finally the de Bruijn factor of the results in Chapter 5 is 4.5.

There are some points of notice: the textual description of all of the above mentioned proofs to some extend contain diagrams to convince the reader of certain steps. Such diagrams are notoriously hard to formalize. Further, the results in Section 7.4 have been our first larger IsaFoR-development and we were still trying to get to grips with Isabelle/HOL. So we probably did not exploit Isabelle/HOL's automatic methods to their full potential, for that reason these proofs are quite verbose, which explains the much higher de Bruijn factor (in comparison to the later developments).

But also the work put in the automatic tool ConCon, although somewhat paling in comparison to the formalization, was significant. ConCon 1.5 consists of approximately 10,000 lines of Scala code and has been very successful in the confluence competition. At the time of writing it won the conditional category three times in a row and is still the only tool in this category to provide certifiable output. Moreover, ConCon can decide confluence of roughly 80% of the systems in Cops and certify approximately 92% of these with the help of CeTA.

12.3 Future Work

We think that the work presented above satisfactory discharges the goal to produce a reliable and automatic tool to check confluence of conditional term rewrite systems. Still, there are some open issues:

[1] `http://www.cs.ru.nl/~freek/factor`

Infeasibility. There are 26 problems in Cops which cannot be solved by ConCon 1.5. One of these is system 327 for computing the greatest common divisor of two natural numbers:

$$\begin{aligned}
\mathsf{gcd}(x,x) &\to x & x < 0 &\to \mathsf{false} & 0 - \mathsf{s}(y) &\to 0\\
\mathsf{gcd}(\mathsf{s}(x),0) &\to \mathsf{s}(x) & 0 < \mathsf{s}(y) &\to \mathsf{true} & x - 0 &\to x\\
\mathsf{gcd}(0,\mathsf{s}(y)) &\to \mathsf{s}(y) & \mathsf{s}(x) < \mathsf{s}(y) &\to x < y & \mathsf{s}(x) - \mathsf{s}(y) &\to x - y\\
\mathsf{gcd}(\mathsf{s}(x),\mathsf{s}(y)) &\to \mathsf{gcd}(x-y,\mathsf{s}(y)) \Leftarrow y < x \approx \mathsf{true}\\
\mathsf{gcd}(\mathsf{s}(x),\mathsf{s}(y)) &\to \mathsf{gcd}(\mathsf{s}(x),y-x) \Leftarrow x < y \approx \mathsf{true}
\end{aligned}$$

Because the system is not left-linear Theorem 4.14 is not applicable. It is also not weakly left-linear and its unraveling is not left-linear, so the methods from Chapter 3 are also not applicable. But system 327 is a quasi-decreasing ADCTRS, making Theorem 5.14 applicable in principle. It only remains to show joinability (or infeasibility for that matter) of its CCPs. The CTRS has three CCPs (modulo symmetry) of which we show two (because the conditions of the third are similar):

$$\begin{aligned}
&\mathsf{gcd}(\mathsf{s}(x),y-x) \approx \mathsf{gcd}(x-y,\mathsf{s}(y)) \Leftarrow y < x \approx \mathsf{true},\ x < y \approx \mathsf{true}\\
&\mathsf{gcd}(x-x,\mathsf{s}(x)) \approx \mathsf{s}(x) \Leftarrow x < x \approx \mathsf{true}
\end{aligned}$$

These CCPs are obviously infeasible (y cannot be strictly smaller and strictly greater than x at the same time for the first CCP and x cannot be strictly smaller than itself for the second one). Unfortunately, this cannot be shown by the methods presented in Chapter 7. By using $\mathcal{R}_{\mathsf{u}}$ (for example when approximating non-reachability) we open the door for inconsistencies:

$$\mathsf{s}(0) \overset{*}{\leftarrow} \mathsf{gcd}(\mathsf{s}(0),\mathsf{s}(0)) \overset{*}{\leftarrow} \mathsf{gcd}(\mathsf{s}(\mathsf{s}(0)),\mathsf{s}(0)) \overset{*}{\to} \mathsf{gcd}(0,\mathsf{s}(\mathsf{s}(0))) \overset{*}{\to} \mathsf{s}(\mathsf{s}(0))$$

and thus $\mathsf{gcd}(\mathsf{s}(\mathsf{s}(0)),\mathsf{s}(0)) < \mathsf{gcd}(\mathsf{s}(\mathsf{s}(0)),\mathsf{s}(0)) \to^* \mathsf{s}(0) < \mathsf{s}(\mathsf{s}(0)) \to^* \mathsf{true}$. Consequently, we may substitute $\mathsf{gcd}(\mathsf{s}(\mathsf{s}(0)),\mathsf{s}(0))$ for both x and y to satisfy the conditions of the CCPs.

It seems that there is still a lot of room for improvement with regard to infeasibility checking. So far all of the infeasibility methods employed by ConCon are unconditional and only approximate the conditional rewrite relation. Maybe we could overcome this problem by employing some external tool that takes conditions into account. Since our tree automata techniques are quite successful in showing infeasibility, the tree automata completion tool Timbuk [15, 20], which implements conditional tree automata completion, would be a natural candidate. Another method that could be extended to allow conditions is the symbol transition graph. This should be investigated further. In general, it would make sense to design, implement and (hopefully) verify a dedicated reachability checker for (conditional) term rewriting. The decomposition of reachability problems from Section 7.3 gives a nice modular framework for that. ConCon is hardly the only tool that could benefit from such a checker.

Other Flavors. For the time being ConCon only supports oriented CTRSs. In principle it should be possible to extend most of the used methods for join CTRSs. Moreover, when employing conditional linearization [12] in order to show the unique normal form property with respect to conversions for non-left-linear TRSs, methods for semi-equational CTRSs are needed and could be implemented in future releases.

Strategy Language. Although ConCon already provides some facilities to set up which methods to use, there are still quite some parts that are hard-coded. Better configurability could for example be beneficial for inlining, which appears to be good to have for our non-confluence methods, but less so for our confluence methods. A kind of strategy language (similar to the one employed by T_TT_2 for example) would be a very nice future feature for ConCon.

Certification. The main reason for the gap of eight systems between the problems that ConCon can show confluent and the ones that CeTA can certify is due to Waldmeister. Unfortunately, although this theorem prover provides some output, it is not detailed enough to be certified, because we cannot reconstruct all of the inferences that were done internally. Providing detailed output (preferably in a certifier-friendly) format should not be too difficult for someone who has already implemented the gory details of a specific method. With some luck the method has been formalized and is already certifiable, even if it is not, a widly-used tool that already provides enough details to certify it in principle, surely gives an incentive for the certifying community to do something about it. I do think that tool authors in general should be made aware of these issues.

Publications

The following publications were produced during the course of my PhD studies (in order of their appearance).

- Cezary Kaliszyk and Thomas Sternagel. Initial experiments on deriving a complete HOL simplification set. In *Proceedings of the 3rd International Workshop on Proof Exchange for Theorem Proving*, volume 14 of *EPiC Series in Computing*, pages 77–86. EasyChair, 2013.
- Thomas Sternagel. KBCV 2.0 - automatic completion experiments. In *Proceedings of the 2nd International Workshop on Confluence*, pages 53–57, 2013. arXiv:1505.01338.
- Thomas Sternagel and Aart Middeldorp. Conditional confluence (system description). In *Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications*, volume 8560 of *Lecture Notes in Computer Science*, pages 456–465. Springer, 2014. doi:10.1007/978-3-319-08918-8_31.
- Thomas Sternagel, Sarah Winkler, and Harald Zankl. Recording Completion for Certificates in Equational Reasoning. In *Proceedings of the 4th ACM-SIGPLAN Conference on Certified Programs and Proofs*, pages 41–47. ACM, 2015. doi:10.1145/2676724.2693171.
- Thomas Sternagel and Aart Middeldorp. Infeasible conditional critical pairs. In *Proceedings of the 4th International Workshop on Confluence*, 2015.
- Christian Sternagel and Thomas Sternagel. Level-confluence of 3-CTRSs in Isabelle/HOL. In *Proceedings of the 4th International Workshop on Confluence*, 2015. arXiv:1602.07115.
- Cynthia Kop, Aart Middeldorp, and Thomas Sternagel. Conditional Complexity. In *Proceedings of the 26th International Conference on Rewriting Techniques and Applications*, volume 36 of *Leibniz International Proceedings in Informatics*, pages 223–240. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.RTA.2015.223.
- Christian Sternagel and Thomas Sternagel. Certifying Confluence of Almost Orthogonal CTRSs via Exact Tree Automata Completion. In *Proceedings of the 1st International Conference on Formal Structures for Computation and Deduction*, volume 51 of *Leibniz International Proceedings in Informatics*, pages 29:1–29:16. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.FSCD.2016.29.
- Thomas Sternagel and Christian Sternagel. Formalized confluence of quasi-decreasing, strongly deterministic conditional TRSs. In *Proceedings of the 5th International Workshop on Confluence*, 2016. arXiv:1609.03341.
- Thomas Sternagel and Christian Sternagel. A characterization of quasi-decreasingness. In *Proceedings of the 15th International Workshop on Termination*, 2016. arXiv:1609.

03345.

- Cynthia Kop, Aart Middeldorp, and Thomas Sternagel. Conditional Complexity *Logical Methods in Computer Science*, 13:1–56, 2017. doi:10.23638/LMCS-13(1:6)2017.
- Christian Sternagel and Thomas Sternagel. Certifying Confluence of Quasi-Decreasing Strongly Deterministic Conditional Term Rewrite Systems. In *Proceedings of the 26th International Conference on Automated Deduction*, volume 10395 of *Lecture Notes in Computer Science*, pages 413–431, Springer, 2017. doi:10.1007/978-3-319-63046-5_26.
- Thomas Sternagel and Christian Sternagel. Certified Non-Confluence with ConCon 1.5. In *Proceedings of the 6th International Workshop on Confluence*, 2017. To appear.

Bibliography

[1] B. Alarcón, R. Gutiérrez, S. Lucas, and R. Navarro-Marset. Proving Termination Properties with MU-TERM. In *Proceedings of the 13th International Conference on Algebraic Methodology and Software Technology*, volume 6486 of *Lecture Notes in Computer Science*, pages 201–208. Springer, 2010. doi:10.1007/978-3-642-17796-5_12.

[2] S. Antoy, B. Braßel, and M. Hanus. Conditional Narrowing without Conditions. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, pages 20–31. ACM Press, 2003. doi:10.1145/888251.888255.

[3] J. Avenhaus and C. Loría-Sáenz. On conditional rewrite systems with extra variables and deterministic logic programs. In *Proceedings of the 5th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 822 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 1994. doi:10.1007/3-540-58216-9_40.

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

[5] E. Balland, Y. Boichut, T. Genet, and P.-E. Moreau. Towards an Efficient Implementation of Tree Automata Completion. In *Proceedings of the 12th International Conference on Algebraic Methodology and Software Technology*, volume 5140 of *Lecture Notes in Computer Science*, pages 67–82. Springer, 2008. doi:10.1007/978-3-540-79980-1_6.

[6] J. Bergstra and J. Klop. Conditional rewrite rules: Confluence and termination. *Journal of Computer and System Sciences*, 32(3):323–362, 1986. doi:10.1016/0022-0000(86)90033-4.

[7] J. C. Blanchette, C. Kaliszyk, L. C. Paulson, and J. Urban. Hammering towards QED. *Journal of Formalized Reasoning*, 9(1):101–148, 2016. doi:10.6092/issn.1972-5787/4593.

[8] J. C. Blanchette and L. C. Paulson. Hammering away – a user's guide to sledgehammer for Isabelle/HOL, 2010. https://isabelle.in.tum.de/dist/doc/sledgehammer.pdf.

[9] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on:

https://gforge.inria.fr/frs/download.php/10994/tata.pdf, 2008. release November, 18th 2008.

[10] Cops: The confluence problems database. http://cops.uibk.ac.at/?q=ctrs.

[11] T. Şerbănuţă and G. Roşu. Computationally equivalent elimination of conditions. In *Proceedings of the 17th International Conference on Rewriting Techniques and Applications*, volume 4098 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2006. doi:10.1007/11805618_3.

[12] R. de Vrijer. Conditional linearization. *Indagationes Mathematicae*, 10(1):145 – 159, 1999. doi:10.1016/S0019-3577(99)80012-3.

[13] N. Dershowitz, M. Okada, and G. Sivakumar. Confluence of conditional rewrite systems. In *Proceedings of the 1st International Workshop on Conditional and Typed Rewriting Systems*, volume 308 of *Lecture Notes in Computer Science*, pages 31–44. Springer, 1988. doi:10.1007/3-540-19242-5_3.

[14] N. Dershowitz and D. A. Plaisted. Rewriting. In J. A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 535–610. Elsevier and MIT Press, 2001.

[15] G. Feuillade and T. Genet. Reachability in conditional term rewriting systems. In *Proceedings of the 4th International Workshop on First-Order Theorem Proving*, volume 86 of *Electronic Notes in Theoretical Computer Science*, pages 133–146, 2003. doi:10.1016/S1571-0661(04)80658-3.

[16] G. Feuillade, T. Genet, and V. Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *Journal of Automated Reasoning*, 33(3-4):341–383, 2004. doi:10.1007/s10817-004-6246-0.

[17] J.-M. Gaillourdet, T. Hillenbrand, B. Löchner, and H. Spies. The new Waldmeister loop at work. In *Proceedings of the 19th International Conference on Automated Deduction*, volume 2741 of *Lecture Notes in Artificial Intelligence*, pages 317–321. Springer, 2003.

[18] H. Ganzinger. Order-sorted completion: the many-sorted way. *Theoretical Computer Science*, 89(1):3–32, 1991. doi:10.1016/0304-3975(90)90105-Q".

[19] T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In *Proceedings of the 9th International Conference on Rewriting Techniques and Applications*, volume 1379 of *Lecture Notes in Computer Science*, pages 151–165. Springer, 1998. doi:10.1007/BFb0052368.

[20] T. Genet and V. Viet Triem Tong. Reachability analysis of term rewriting systems with timbuk. In *Proceedings of the 8th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, volume 2250 of *Lecture Notes in Computer Science*, pages 695–706. Springer, 2001. doi:10.1007/3-540-45653-8_48.

[21] J. Giesl, M. Brockschmidt, F. Emmes, F. Frohn, C. Fuhs, C. Otto, M. Plücker, P. Schneider-Kamp, T. Ströder, S. Swiderski, and R. Thiemann. Proving Termination of Programs Automatically with AProVE. In *Proceedings of the 7th International Joint Conference on Automated Reasoning*, volume 8562 of *Lecture Notes in Computer Science*, pages 184–191. Springer, 2014. doi:10.1007/978-3-319-08587-6_13.

[22] J. Giesl, R. Thiemann, and P. Schneider-Kamp. Proving and disproving termination of higher-order functions. In *Proceedings of the 5th International Workshop on Frontiers of Combining Systems*, volume 3717 of *Lecture Notes in Computer Science*, pages 216–231. Springer, 2005. doi:10.1007/11559306_12.

[23] K. Gmeiner. *Transformational Approaches for Conditional Term Rewrite Systems*. PhD thesis, Vienna PhD School of Informatics, TU Wien, 2014.

[24] K. Gmeiner. CoScart: Confluence prover in Scala. In *Proceedings of the 5th International Workshop on Confluence*, page 82, 2016.

[25] K. Gmeiner, B. Gramlich, and F. Schernhammer. On (Un)Soundness of Unravelings. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, volume 6 of *Leibniz International Proceedings in Informatics*, pages 119–134. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2010. doi:10.4230/LIPIcs.RTA.2010.119.

[26] K. Gmeiner and N. Nishida. Notes on Structure-Preserving Transformations of Conditional Term Rewrite Systems. In *Proceedings of the 1st International Workshop on Rewriting Techniques for Program Transformations and Evaluation*, volume 40 of *OpenAccess Series in Informatics*, pages 3–14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2014. doi:10.4230/OASIcs.WPTE.2014.3.

[27] K. Gmeiner, N. Nishida, and B. Gramlich. Proving confluence of conditional term rewriting systems via unravelings. In *Proceedings of the 2nd International Workshop on Confluence*, pages 35–39, 2013.

[28] M. Hanus. On extra variables in (equational) logic programming. In *Proceedings of the 12th International Conference on Logic Programming*, pages 665–679. MIT Press, 1995.

[29] G. Huet. Confluent reductions: Abstract properties and applications to term rewriting systems. *J. ACM*, 27(4):797–821, 1980.

[30] T. Ida and S. Okui. Outside-in conditional narrowing. *IEICE Transactions on Information and Systems*, E77-D(6):631–641, 1994.

[31] F. Jacquemard. Decidable approximations of term rewriting systems. In *Proceedings of the 7th International Conference on Rewriting Techniques and Applications*, volume 1103 of *Lecture Notes in Computer Science*, pages 362–376. Springer, 1996. doi:10.1007/3-540-61464-8_65.

[32] C. Kop, A. Middeldorp, and T. Sternagel. Conditional Complexity. In *Proceedings of the 26th International Conference on Rewriting Techniques and Applications*, volume 36 of *Leibniz International Proceedings in Informatics*, pages 223–240. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2015. doi:10.4230/LIPIcs.RTA.2015.223.

[33] C. Kop, A. Middeldorp, and T. Sternagel. Conditional Complexity. *Logical Methods in Computer Science*, 13:1–56, 2017. doi:10.23638/LMCS-13(1:6)2017.

[34] M. Korp, C. Sternagel, H. Zankl, and A. Middeldorp. Tyrolean Termination Tool 2. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications*, volume 5595 of *Lecture Notes in Computer Science*, pages 295–304. Springer, 2009. doi:10.1007/978-3-642-02348-4_21.

[35] E. Kounalis and M. Rusinowitch. A proof system for conditional algebraic specifications. In *Proceedings of the 2nd International Workshop on Conditional and Typed Rewriting Systems*, volume 516 of *Lecture Notes in Computer Science*, pages 51–63. Springer, 1991. doi:10.1007/3-540-54317-1_80.

[36] S. Lucas. Context-sensitive computations in functional and functional logic programs. *The Journal of Functional and Logic Programming*, 1998(1), 1998.

[37] S. Lucas, C. Marché, and J. Meseguer. Operational termination of conditional term rewriting systems. *Inf. Process. Lett.*, 95(4):446–453, 2005. doi:10.1016/j.ipl.2005.05.002.

[38] S. Lucas and J. Meseguer. Dependency pairs for proving termination properties of conditional term rewriting systems. *The Journal of Logic and Algebraic Methods in Programming*, 86(1):236–268, 2017. doi:10.1016/j.jlamp.2016.03.003.

[39] S. Lucas, J. Meseguer, and R. Gutiérrez. Extending the 2D Dependency Pair Framework for Conditional Term Rewriting Systems. In *Proceedings of the 24th International Symposium on Logic-Based Program Synthesis and Transformation*, volume 8981 of *Lecture Notes in Computer Science*, pages 113–130. Springer, 2014. doi:10.1007/978-3-319-17822-6_7.

[40] M. Marchiori. On deterministic conditional rewriting. Computation Structures Group Memo 405, MIT Laboratory for Computer Science, 1987.

[41] M. Marchiori. Unravelings and ultra-properties. In *Proceedings of the 5th International Conference on Algebraic and Logic Programming*, volume 1139 of *Lecture Notes in Computer Science*, pages 107–121, 1996. doi:10.1007/3-540-61735-3_7.

[42] A. Middeldorp. Approximating dependency graphs using tree automata techniques. In *Proceedings of the 1st International Joint Conference on Automated Reasoning*, volume 2083 of *Lecture Notes in Artificial Intelligence*, pages 593–610, 2001. doi:10.1007/3-540-45744-5_49.

[43] A. Middeldorp and E. Hamoen. Completeness Results for Basic Narrowing. *Applicable Algebra in Engineering, Communication and Computing*, 5:213–253, 1994. doi:10.1007/BF01190830.

[44] J. Nagele, B. Felgenhauer, and A. Middeldorp. CSI: New evidence – a progress report. In *Proceedings of the 26th International Conference on Automated Deduction*, volume 10395 of *Lecture Notes in Computer Science*, pages 385–397. Springer, 2017. doi:10.1007/978-3-319-63046-5_24.

[45] M. Newman. On theories with a combinatorial definition of equivalence. *Annals of Mathematics*, 43(2):223–243, 1942. doi:10.2307/1968867.

[46] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. doi:10.1007/3-540-45949-9.

[47] N. Nishida, T. Kuroda, and K. Gmeiner. CO3 version 1.3. In *Proceedings of the 5th International Workshop on Confluence*, page 79, 2016.

[48] N. Nishida, M. Sakai, and T. Sakabe. Soundness of unravelings for conditional term rewriting systems via ultra-properties related to linearity. *Logical Methods in Computer Science*, 8:1–49, 2012. doi:10.2168/LMCS-8(3:4)2012.

[49] E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002.

[50] F. Schernhammer and B. Gramlich. VMTL - a modular termination laboratory. In *Proceedings of the 20th International Conference on Rewriting Techniques and Applications*, volume 5595 of *Lecture Notes in Computer Science*, pages 285–294. Springer, 2009. doi:10.1007/978-3-642-02348-4_20.

[51] F. Schernhammer and B. Gramlich. Characterizing and proving operational termination of deterministic conditional term rewriting systems. *The Journal of Logic and Algebraic Programming*, 79(7):659–688, 2010. doi:10.1016/j.jlap.2009.08.001.

[52] C. Sternagel and T. Sternagel. Level-confluence of 3-CTRSs in Isabelle/HOL. In *Proceedings of the 4th International Workshop on Confluence*, pages 28–32, 2015. arXiv:1602.07115.

[53] C. Sternagel and T. Sternagel. Certifying Confluence of Almost Orthogonal CTRSs via Exact Tree Automata Completion. In *Proceedings of the 1st International Conference on Formal Structures for Computation and Deduction*, volume 51 of *Leibniz International Proceedings in Informatics*, pages 29:1–29:16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.FSCD.2016.29.

[54] C. Sternagel and T. Sternagel. Certifying Confluence of Quasi-Decreasing Strongly Deterministic Conditional Term Rewrite Systems. In *Proceedings of the 26th International Conference on Automated Deduction*, volume 10395 of *Lecture Notes in Computer Science*, pages 413–431. Springer, 2017. doi:10.1007/978-3-319-63046-5_26.

[55] C. Sternagel and R. Thiemann. Signature extensions preserve termination - an alternative proof via dependency pairs. In *Proceedings of the 19h International Workshop on Computer Science Logic*, volume 6247 of *Lecture Notes in Computer Science*, pages 514–528. Springer, 2010. doi:10.1007/978-3-642-15205-4_39.

[56] C. Sternagel and R. Thiemann. The Certification Problem Format. In *Proceedings of the 11th Workshop on User Interfaces for Theorem Provers*, pages 61–72, 2014. doi:10.4204/EPTCS.167.8.

[57] T. Sternagel. KBCV 2.0 - automatic completion experiments. In *Proceedings of the 2nd International Workshop on Confluence*, pages 53–57, 2013. arXiv:1505.01338.

[58] T. Sternagel and A. Middeldorp. Conditional confluence (system description). In *Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications*, volume 8560 of *Lecture Notes in Computer Science*, pages 456–465. Springer, 2014. doi:10.1007/978-3-319-08918-8_31.

[59] T. Sternagel and A. Middeldorp. Infeasible conditional critical pairs. In *Proceedings of the 4th International Workshop on Confluence*, pages 13–17, 2015.

[60] T. Sternagel and C. Sternagel. Formalized confluence of quasi-decreasing, strongly deterministic conditional TRSs. In *Proceedings of the 5th International Workshop on Confluence*, pages 60–64, 2016. arXiv:1609.03341.

[61] T. Sternagel and C. Sternagel. Certified non-confluence with ConCon 1.5. In *Proceedings of the 6th International Workshop on Confluence*, 2017. To appear.

[62] T. Sternagel, S. Winkler, and H. Zankl. Recording Completion for Certificates in Equational Reasoning. In *Proceedings of the 4th ACM-SIGPLAN Conference on Certified Programs and Proofs*, pages 41–47. ACM Press, 2015. doi:10.1145/2676724.2693171.

[63] T. Sternagel and H. Zankl. KBCV - Knuth-Bendix completion visualizer. In *Proceedings of the 6th International Joint Conference on Automated Reasoning*, volume 7364 of *Lecture Notes in Artificial Intelligence*, pages 530–536. Springer, 2012. doi:10.1007/978-3-642-31365-3_41.

[64] T. Suzuki, A. Middeldorp, and T. Ida. Level-confluence of conditional rewrite systems with extra variables in right-hand sides. In *Proceedings of the 6th International Conference on Rewriting Techniques and Applications*, volume 914 of *Lecture Notes in Computer Science*, pages 179–193. Springer, 1995. doi:10.1007/3-540-59200-8_56.

[65] R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468. Springer, 2009. doi:10.1007/978-3-642-03359-9_31.

[66] S. Winkler and R. Thiemann. Formalizing soundness and completeness of unravelings. In *Proceedings of the 10th International Workshop on Frontiers of Combining Systems*, volume 9322 of *Lecture Notes in Computer Science*, pages 239–255. Springer, 2015. doi:10.1007/978-3-319-24246-0_15.

[67] A. Yamada, K. Kusakari, and T. Sakabe. Nagoya Termination Tool. In *Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications*, volume 8560 of *Lecture Notes in Computer Science*, pages 466–475. Springer, 2014. doi:10.1007/978-3-319-08918-8_32.

[68] H. Zankl, B. Felgenhauer, and A. Middeldorp. CSI – A confluence tool. In *Proceedings of the 23rd International Conference on Automated Deduction*, volume 6803 of *Lecture Notes in Artificial Intelligence*, pages 499–505. Springer, 2011. doi:10.1007/978-3-642-22438-6_38.

[69] H. Zankl and A. Middeldorp. Equational reasoning for termination of rewriting. In *Proceedings of the 10th International Workshop on Termination*, pages 112–115, 2009.

Index

Page numbers in **boldface** refer to definitions.

Symbols

A

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V