Automatic Certification of Termination Proofs

dissertation

by

Christian 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 science

advisor: *Univ.-Prof. Dr. Aart Middeldorp, Informatik*

Innsbruck, *August 10, 2010*

Automatic Certification of Termination Proofs

Automatic Certification of Termination Proofs

Christian Sternagel

August 10, 2010

This document was prepared with LaTeX using the KOMA-Script packages.

for mikro

Preface

What you hold in hand—my doctoral thesis—is the result of about four hard/exciting/frustrating/enjoyable years of work. When I started back in 2006, I had the title of my thesis and the ambition of formalizing at least the dependency pair transformation and the subterm criterion. What I did not have, was any real experience in using Isabelle/HOL (or any other interactive theorem prover, for that matter). It turned out that coming to grips with Isabelle took quite some time, especially, since there where many decisions when designing IsaFoR, which had a high impact on the smoothness of later formalizations. Not all of them where made for the better in the first run. So, I spent a lot of time on the basic layer about abstract rewriting and term rewriting and did rewrite everything several times. At some point—after this initial phase reached a fixed point and my co-developer, René Thiemann, joined—the project suddenly gained momentum. This was also, when we settled on code-generation for obtaining an efficient certifier. Soon, our combined efforts resulted in the first version (0.99) of CeTA in March 2009. From that point onwards, CeTA's release cycle was quite tight. Until, in the current version (1.15; released in August 2010), most of the powerful termination techniques that are used by termination tools, where supported by CeTA.

The development of IsaFoR/CeTA took place partly in parallel with preparing my thesis, during which I adapted to two releases of Isabelle/HOL and at least five releases of CeTA. First, I thought that most of the work was done after finishing all the formalizations in Isabelle. Far from it! Just because you have proven something, does not mean that you have a nicely structured and easily understandable proof. Thus, I put a lot of work into rewriting existing proofs, just to make them better structured and clearer. Sometimes, I could even remove large parts of proofs (mostly concerned with technical details), since at the time of first proving, I just did not exploit Isabelle's automatic methods to their full potential.

Additionally, I decided that all of the proofs which are part of my thesis, should be checked automatically, using Isabelle's document preparation features. At the same time, I wanted to use notation that is also used in rewriting literature. To achieve both of these goals together, I finally ended up, entering 'gibberish' in the sources of my thesis. To demonstrate what I mean, here is what I had to type to get a nicely typeset and automatically checked proof of Lemma 4.1 (which is one of the shortest proofs of our formalization):

```
lemma not_wf_trs_imp_not_SN_rstep:
  assumes "\<not> \<^raw:\wftrs{>\<R>\<^raw:}>"
  shows "\<not> \<^raw:\SNrstep{>\<R>\<^raw:}>"
proof -
  from assms obtain l r where "(l, r) \<in> \<R>"
    and bad_rule: "\<^raw:\isvar{>l\<^raw:}> \<or> (\<exists>x.
      x \<in> \<^raw:\varsterm{>r\<^raw:}>
        -\<^raw:\varsterm{>l\<^raw:}>)"
      unfolding wf_trs_def' by auto
  from bad_rule show ?thesis
  proof
    assume "\<^raw:\isvar{>l\<^raw:}>"
    then obtain x where l: "l = Var x" by (cases l) simp_all
    (*<*)
    let ?\<sigma> = "\<^raw:\subst{>x\<^raw:}{>r\<^raw:}>"
    (*>*)
    let ?S = "\<lambda>i. \<^raw:\subapp{>(Var x)\<^raw:}{>(
      \<^raw:\subst{>x\<^raw:}{>r\<^raw:}>
        \<^raw:\pow{>i\<^raw:}>)\<^raw:}>"
    have "\<forall>i. (?S i) \<^raw:\rstep[>\<R>\<^raw:]>
      (?S (\<^raw:\suc{>i\<^raw:}>))"
    proof%fold
      fix i
      from rstep.subst[OF rstep.id[OF `(l, r) \<in> \<R>`],
        of "?\<sigma>\<^raw:\pow{>i\<^raw:}>" ]
        show "(?S i) \<^raw:\rstep[>\<R>\<^raw:]>
          (?S (\<^raw:\suc{>i\<^raw:}>))"
        by (simp add: l(*<*)subst_def(*>*))
    qed
    thus ?thesis by best
  next
    assume "\<exists>x.
      x \<in> \<^raw:\varsterm{>r\<^raw:}>
```

```
      -\<^raw:\varsterm{>l\<^raw:}>"
    then obtain x where "x \<in> \<^raw:\varsterm{>r\<^raw:}>
      -\<^raw:\varsterm{>l\<^raw:}>"
      (*<*)and empty: "vars_term l \<inter> {x} = {}"(*>*)
      by auto
    hence "r \<unrhd> Var x" by (induct r) auto
    then obtain C where (*<*)r:(*>*)"r =
      \<^raw:\ctxtapp{>C\<^raw:}{>Var x\<^raw:}>"
      by (rule supteqp_ctxt_E)
    (*<*)
    let ?\<sigma> = "\<^raw:\subst{>x\<^raw:}{>l\<^raw:}>"
    (*>*)
    let ?S = "\<lambda>i. \<^raw:\ctxtapp{>(
      (\<^raw:\subappctxt{>C\<^raw:}{>
      \<^raw:\subst{>x\<^raw:}{>l\<^raw:}>\<^raw:}>)
      \<^raw:\pow{>i\<^raw:}>)\<^raw:}{>l\<^raw:}>"
    (*<*)
    from subst_apply_id'[OF `vars_term l \<inter> {x} = {}`,
      of "?\<sigma>"]
      have l: "\<^raw:\subapp{>l\<^raw:}{>
        ?\<sigma>\<^raw:}> = l" by simp
    (*>*)
    have "\<forall>i. (?S i) \<^raw:\rstep[>\<R>\<^raw:]>
      (?S (\<^raw:\suc{>i\<^raw:}>))"
    proof%fold
      fix i
      from rstepI[OF `(l, r) \<in> \<R>`,
        of _ "(C \<cdot>c ?\<sigma>)^i" ?\<sigma>]
        have "(((C \<cdot>c ?\<sigma>)^i)\<langle>l\<rangle>,
          ((C \<cdot>c ?\<sigma>)^i \<circ>c
          (C \<cdot>c ?\<sigma>)^Suc 0)
          \<langle>l\<rangle>) \<in> rstep \<R>"
        unfolding l r subst_apply_term_ctxt_apply_distrib
        by (simp add: subst_def)
      thus "(?S i, ?S (Suc i)) \<in> rstep \<R>"
      unfolding ctxt_power_compose_distr[symmetric] by simp
    qed
    thus ?thesis by best
  qed
qed
```

A final word of warning: theorem proving is addictive.

Acknowledgments

I am grateful to my supervisor Aart Middeldorp, who encouraged me towards interactive theorem proving in the first place.

On my way of learning Isabelle, I am especially indebted to Alexander Krauss, a never-ending source of answers to my many questions about Isabelle. Further, my thank goes to the Isabelle developers and the whole Isabelle community, both are very kind to newcomers and anxious to answer questions and solve problems.

Among my colleagues in Innsbruck, I am especially grateful to Georg Moser and Harald Zankl, who had to convince me several times that I would be able to manage my formalizations in time. Harald deserves a second mention, for providing many nice 'Reindlings' and pizze.

My co-developer René Thiemann was irreplaceable in pushing CeTA. Without him, CeTA would support less than half of the techniques that it does now.

It goes without saying that my family and friends have been crucial in my life and thus have also influenced this work. I would like to thank them all for the support given during the preparation of my thesis, especially my companion in life, Nicola Blassnig (mikro), who had to bear me stressed out and doubtful.

Finally, I would like to thank the Austrian Science Fund, which supported my research through the FWF project P18763.

Christian Sternagel

Innsbruck, Austria
August 10, 2010

Contents

Chapter 1

Introduction

The man of science has learned to believe in
justification, not by faith, but by verification.

Thomas Henry Huxley
On the Advisableness of Improving Natural Knowledge

Proving the correctness of computer software is of utmost importance for safety-critical systems (like fire alarms, fly-by-wire systems, human spaceflight, nuclear reactors, robotic surgery, etc.). A crucial part of proving correctness is to show that a computer program always yields a result, as it may run forever otherwise. This property is called *termination* and is undecidable in general. Considering the plethora of existing programming languages as well as the rate at which new languages arise, basic facts about programming languages are usually handled on a more abstract level, using some mathematical model of computation instead of a specific programming language. This facilitates mathematical reasoning and induces facts that are applicable to all concrete implementations. We use *term rewriting* as model of computation.

In term rewriting, we consider *term rewrite systems*, instead of programs. A term rewrite system is a set of simple rules that describes how to compute a result from some given input. There are already several transformations, such that termination of a program (written in a specific programming language), follows from the termination of a term rewrite system that has been generated from this program. Hence, termination of term rewrite systems is of practical use.

Now, the question arises: How do we prove termination of a given term rewrite system? This has been a topic of research, for several decades. Many so called

termination techniques have been developed. Some of them are convenient for proving termination by hand, others are especially suitable for automation. Most of these techniques have been integrated in at least one of several automated *termination tools*. These tools are programs that take a term rewrite system and try to either prove or disprove termination. Aside from the long (at least for computer science) history of termination analysis, there are still recent and powerful methods for proving termination automatically. Thus, by using a termination tool it is sometimes possible to prove termination of a term rewrite system at the touch of a button. This, on the other hand, may give rise to an automatic correctness proof for a given computer program and thus increases software safety in general.

But wait a minute! We are trying to prove correctness of a program by using another program: the termination tool. How do we know that the termination tool is correct? The short answer is: we do not. Indeed, there have already been some occasions, where a termination tool delivered a wrong proof. This is not too surprising, since the average termination tool is a complex piece of software that has to combine many different termination techniques in an efficient way. Combining many different techniques additionally results in huge proofs: modern termination tools may easily generate proofs reaching sizes of several megabytes. That means, a human would have to read hundreds of pages of text, in order to check the correctness of such a proof. This is clearly insufficient to reliably check the correctness of termination proofs. There are mainly two approaches of avoiding manual checking and still maintaining a high level of confidence in the correctness of the generated proofs: The first approach is to formally verify the correctness of a termination tool. The main disadvantage being that whenever the tool is modified (to make it more efficient or more powerful by adding a new termination technique, say), the correctness proof may break. The second approach is to implement a trustworthy *termination proof checker* that reads a given termination proof and checks its correctness automatically. The main advantage being that termination tools do not have to be modified much. It suffices to generate proofs in a format that can be read by the proof checker. We take the second alternative.

The remaining obstacles are: having a common proof format and developing a trustworthy termination proof checker. We know of three projects that aim at checking termination proofs automatically (including our own). The first obstacle has been settled in cooperation of all three projects by introduc-

ing a common XML format for termination proofs.[1] The second obstacle is generally tackled by using interactive *theorem provers*. In our project we use Isabelle/HOL.

An interactive theorem prover allows to conduct computer-aided mathematical proof. This way of proving is akin to a proof on paper, but with two main differences: Every step in the proof is machine-checked. And often, we need to provide much more detail than in a paper-proof in order to 'convince' the theorem prover of the correctness of our workings. In the following, we mean computer-aided mathematical proof, when we speak about *formalizing* something. The results of formalizing theorems are highly reliable mathematical proofs. Additionally, some theorem provers (like Isabelle/HOL) provide facilities to extract programs from formalized proofs. Such programs are correct by construction and are thus much more trustworthy than handcrafted code. Just to be sure, we will never reach a 100% certainty that some program is correct. It is however important to minimize the remaining uncertainty as far as possible. There is a nice motivation by Dijkstra (1970), given in his *Notes on Structured Programming*:

> *If the chance of correctness of an individual component equals p, the chance of correctness of a whole program, composed of N such components, is something like $P = p^N$. As N will be very large, p should be very, very close to* 1 *if we desire P to differ significantly from zero!*

So if we want to trust a termination proof, composed of many different termination techniques, we better reach a high assurance that each involved technique is applied correctly (and correct on its own).

Our termination proof checker is built in two stages:

- First, we formalize all the mathematical theory that is used in automated termination tools. Additionally, we formalize functions that check, whether some termination technique is applied correctly. This is done in our ***Isa****belle* ***F****ormalization* ***o****f* ***R****ewriting*.
- Then, we extract the termination proof checker using Isabelle's code-generation facilities. Resulting in our ***Ce****rtified* ***T****ermination* ***A****nalyser*.

[1] `http://cl-informatik.uibk.ac.at/software/cpf`

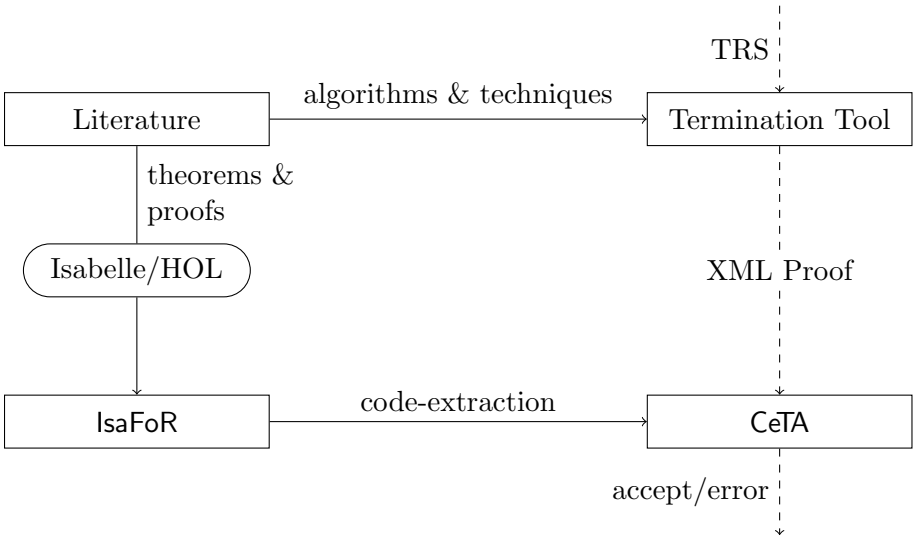

Figure 1.1: The General Picture

The general picture is depicted in Figure 1.1, where solid lines are used for entities that are fixed once and for all (like theorems about termination techniques) and dashed lines denote inputs and outputs that vary (like a specific TRS). Termination tools implement termination techniques and algorithms that are described in the literature. For certification, we formalize theorems about those techniques and algorithms in Isabelle/HOL. On the result, we apply Isabelle's code-extractor to obtain our certifier. Which, in turn, is run on a termination proof that was generated by some termination tool on some TRS. The final outcome is a fully verified answer, whether the generated proof is correct. In case it is not, we obtain an informative error message.

In the following we indicate our specific contributions.

1.1 Contributions

As already stated above, the IsaFoR/CeTA project mainly arose out of the need to automatically check termination proofs that are generated by termination

tools. Additionally, we wanted to give an Isabelle formalization of term rewriting that can be used as the bases for related projects. Our main contributions to IsaFoR/CeTA are as follows:

Dependency Pair Transformation We formalized the dependency pair transformation and the corresponding theorem, stating that the termination of a given term rewrite system is equivalent to finiteness of the resulting dependency pair problem.

Subterm Criterion We formalized the subterm criterion, a termination technique for the dependency pair framework. This also served as the basis for a variant of the size-change principle that can be used to prove finiteness of dependency pair problems.

Signature Extensions Building on the dependency pair transformation, we gave an alternative (and in our opinion elegant) proof of the fact that signature extensions of term rewrite systems preserve termination. Additionally, we first proved and formalized that for (a certain class of) dependency pair problems, signature restrictions reflect (minimal) infinite chains.

Root-Labeling We formalized root-labeling, a transformation on term rewrite systems as well as dependency pair problems, based on semantic labeling. However, in most cases root-labeling can only be applied after another transformation, the so called *closure under flat contexts*. This is strongly related to signature extensions.

Furthermore, we formalized large parts of the basic infrastructure available in IsaFoR. Starting from abstract rewriting, via terms, contexts, positions, substitutions, and so on, up to the dependency pair framework (today's de facto standard for modular termination proofs). Finally, we contributed to CeTA specific developments, like providing informative error messages, in the case that a proof is not accepted. These last two activities have been done in strong collaboration with my co-developer and hence are not described in the thesis.

The following section describes where to find more detailed descriptions of our contributions and the IsaFoR/CeTA internals in general.

1.2 Overview

Since large parts of our work are concerned with formalizing facts about rewriting using a theorem prover, there are mainly two audiences for our work: researches in term rewriting and researches in theorem proving. Additionally, we want to stay self-contained, as far a possible. Hence, we will give all the necessary basics of term rewriting that are used in our formalizations (also, since those basics form a large part of those formalizations). Furthermore, we try to provide some basics in theorem proving using Isabelle/HOL. We hope that this proves interesting for both audiences.

The outline of our thesis is as follows: We start by a short introduction to interactive theorem proving in Chapter 2. Thus, well equipped, we present our formalization of abstract rewriting in Chapter 3. This includes a proof of Newman's Lemma, to get further acquainted with Isabelle. On top of abstract rewriting, we formalize (first-order) term rewriting in Chapter 4. Next, comes the leap from termination of term rewrite systems to finiteness of dependency pair problems; provided by the dependency pair transformation, formalized in Chapter 5. This is a crucial part of our formalizations, since it paves the way to the dependency pair framework, a prerequisite of many termination techniques that are available in CeTA. Inside the dependency pair framework, so called processors are used to simplify problems recursively. In Chapter 6, we give our formalization of the subterm criterion processor. Afterwards, in Chapter 7, we show how the dependency pair transformation can be used to prove facts about signature extensions that are itself necessary for the formalization of several termination techniques. One of those, namely root-labeling, is handled in Chapter 8. In Chapter 9, we show how computable check functions are used to code-extract the efficient termination proof checker CeTA. Finally, we conclude in Chapter 10.

1.3 Terminology

Throughout this thesis by *formalizations*, we mean proofs and definitions conducted in an interactive theorem prover. Sometimes, we call such proofs *machine-checked* or even *mechanized*. When we talk about *proof checking* or

certification, we usually mean that we check a termination proof for a given term rewrite system for its correctness. By *code-generation* or *code-extraction*, we denote the process of automatically creating program source files for some programming language from a formalization.

1.4 Using Isabelle/IsaFoR

Throughout the text, we will indicate the most important reasoning patterns for concepts introduced by IsaFoR. This is done inside 'reasoning boxes' of the following shape:

Reasoning about some Concept.
Here we describe how to use Isabelle/IsaFoR to reason about 'some Concept.'

Also note that for better readability, the document introduces so called syntax translations for many defined constants of IsaFoR. This means that what you see in this text, should be close to the usual notation in the literature, but may differ radically from what you would have to type when using IsaFoR in your own Isabelle theories. The 'reasoning boxes' are also the place, where we will point out the ASCII versions of such constants.

1.5 Chapter Notes

At the end of each chapter, we give a short summary, indicate related work, and cite relevant publications from the literature (this avoids breaking the 'narrative flow' in the rest of the chapter).

In this chapter, we gave reasons for analyzing termination of term rewrite systems: the final goal is to prove termination of computer programs, which is an important part of the verification of programs and results in more reliable software. We further motivated our interest in the certification of termination proofs using a theorem prover.

The fact that there is no algorithm that takes some computer program as input and decides whether it is terminating or not, has been proven by Turing [45].

A thorough introduction to term rewriting is given by Baader and Nipkow [2]. The traditional introduction to Isabelle/HOL is the tutorial by Nipkow et al. [29].

Chapter 2

Isabelle/HOL

Beware of bugs in the above code;
I have only proved it correct,
not tried it.

Donald Ervin Knuth

Axiomatic type-classes are definitional.
Type definitions are axiomatic.

Makarius Wenzel
Isabelle Developers Joke

In all our formalizations we use the Isabelle/HOL interactive theorem prover. Hence, it seems appropriate to give a short introduction to automated reasoning in general and interactive theorem proving using Isabelle/HOL in particular. Informally speaking, *automated reasoning* is concerned with aspects of reasoning that may be implemented on a computer. Two subfields are *automated theorem proving* and *automated proof checking*. In automated theorem proving, a computer program is used to automatically find proofs of mathematical theorems, whereas in automated proof checking, a computer program merely checks whether every inference step of a given proof is correct. The combination of these two tasks is *interactive theorem proving*, where some parts of a proof are found automatically, but others have to be filled in by the user in full detail and it is checked by the machine that every step of the user is according to the rules.

More specifically, in order to allow a computer program to check or even find proofs, we need a formal system (*logic*), describing the facts that may be stated (*formulas*) as well as the valid reasoning patterns (*inference rules*) that are

available for deriving new valid facts from given valid facts. This is the general picture. How this is realized in Isabelle/HOL, is discussed in the next section.

2.1 Isabelle - A Generic Theorem Prover

Isabelle itself, is a generic theorem prover. This means that it may be instantiated to specific object-logics (like higher-order logic, in the case of Isabelle/HOL). Isabelle's meta-logic is an intuitionistic fragment of higher-order logic. The meta-logic is usually denoted by Pure (or sometimes Isabelle/Pure).

The most important constructs of Pure are: $\Rightarrow$, $\equiv$, $\bigwedge$, and $\Longrightarrow$. Of those, $\Rightarrow$ is the (right-associative) *function type constructor*, that is, $\alpha \Rightarrow \beta$ denotes the type of functions taking arguments of type α and yielding results of type β. We write $t{::}\tau$, to express that the term t is of type τ. The only primitive type that is available in Pure is *prop*, denoting *propositions*. Any concrete object-logic specifies what constitutes a proposition (for example, in HOL, every term of type *bool* is a proposition). Having types, we can give more information on the other three constructs, namely

$$\begin{aligned} \equiv &:: \alpha \Rightarrow \alpha \Rightarrow \mathit{prop} \\ \bigwedge &:: (\alpha \Rightarrow \mathit{prop}) \Rightarrow \mathit{prop} \\ \Longrightarrow &:: \mathit{prop} \Rightarrow \mathit{prop} \Rightarrow \mathit{prop} \end{aligned}$$

These are the logical connectives of the meta-logic. Here, $\equiv$ denotes meta-level equality. This is for example used to define new constants in terms of existing ones. Further, $\bigwedge$ denotes meta-level universal quantification, for example, $\bigwedge x.\ P\ x$ means that the predicate P is true for every argument x. Proving such statements, would be formulated as proving something about "an arbitrary but fixed x," in a paper-proof. The type of $\bigwedge$ may be somewhat cryptic and hence deserves further explanation. Obviously, we have a higher-order construct, taking a function of type $\alpha \Rightarrow \mathit{prop}$ as argument, where α is the type of the bound variable. In the end, $\bigwedge x.\ P\ x$ is just syntactic sugar for $\bigwedge(\lambda x.\ P\ x)$. The intended meaning is that whatever argument we give to the function $\lambda x.\ P\ x$, the resulting proposition has to be true. Finally, $\Longrightarrow$ denotes (the right-associative) meta-implication, as in $A \Longrightarrow B$, which states that A *implies* B. In a paper-proof, this would correspond to "assuming A we show B." There is syntactic sugar for a chain of meta-implications of the form $A_1 \Longrightarrow ... \Longrightarrow$

$A_n \Longrightarrow B$. All the assumptions are grouped together between double-brackets and are separated by semicolons, resulting in: $[\![A_1; ...; A_n]\!] \Longrightarrow B$.

Using meta-level universal quantification together with meta-level implication, we can express inference rules (that is, the rules we must adhere to, inside the logical setting we choose). For readability, Isabelle drops outermost universal quantification at the meta-level, that is, the inference rule

$$\bigwedge P\ Q\ x.\ P\ x \Longrightarrow Q\ x$$

is equivalent to the inference rule

$$P\ x \Longrightarrow Q\ x$$

As a more realistic example, consider the rule for disjunction elimination from HOL:

$$\mathit{disjE}\colon [\![P \vee Q;\ P \Longrightarrow R;\ Q \Longrightarrow R]\!] \Longrightarrow R$$

This rule states that for arbitrary P and Q, whenever we know that either P or Q holds (denoted by $P \vee Q$ in HOL) and in addition some fact R is derivable from both of P and Q, then we may conclude that the fact R is true.

Reasoning in the Meta-Logic.
The ASCII versions of the meta-logical constructs are => for the function type constructor, == for meta-level equality, !! for meta-level universal quantification, and ==> for meta-level implication. For the double-brackets you have to type [| and |].

2.2 Higher-Order Logic

HOL is an object-logic built on top of Pure. It is an implementation of classical type theory. Furthermore, there is a huge library containing orderings, sets, natural numbers, the Axiom of Choice, lists, maps, strings and what is more. HOL is sometimes described by the informal equation

HOL = Functional Programming + Logic

which gives a rough feeling, what reasoning in HOL is like. In contrast to the Isabelle tutorial—which uses the now obsolete apply-style for proofs—we exclusively do our formalizations using Isar (standing for *Intelligible semi-automated*

reasoning; surely it is just a coincidence that the river of the same name is flowing just a few hundred meters from where Isar was developed). Isar is a (no longer) new style of conducting proofs in Isabelle. A nice feature of Isabelle/Isar is that formally checked theory files can be referenced by LaTeX documents, thereby offering the advantage that every proof and every formula is run through Isabelle in the background. This drastically reduces the chance of errors in the presented material (of course it is still possible that what is presented does not coincide with what is claimed to be presented). Using this feature, also means that every proof is displayed directly in Isar, which (at least by name) should be intelligible. We believe that after a short initial phase, where Isar needs getting used to, this kind of presentation makes the structure of a proof more clear than most mathematical text proofs. Also note that we freely drop details from proofs, where in our opinion they do not contribute to an understanding of the proof (still the details that are omitted from the presentation are present at the level where Isabelle checks our proofs). Parts of proofs, where we do omit details are marked by "⟨*proof omitted*⟩."

2.2.1 Logic

Concerning the *Logic* part of the above equation, Isabelle has the usual connectives: $=$ (*equality*), $\forall$ (*universal quantification*), $\exists$ (*existential quantification*), $\wedge$ (*conjunction*), $\vee$ (*disjunction*), $\neg$ (*negation*), and $\longrightarrow$ (*implication*); as well as the two constants *True* (*truth*) and *False* (*absurdity*).

Reasoning in HOL.
The ASCII versions of these connectives are as follows: =, *ALL*, *EX*, &, |, ~, −−>, *True*, and *False*.

Before we have a look at Isabelle's functional programming features, let us have a look at an Isabelle/Isar/HOL[1] proof. Consider the following Isabelle proof of the well-known fact that a universal quantifier in the conclusion of an implication, may be pulled to the front:

lemma *example1*:
assumes $A \longrightarrow (\forall x.\ P\ x)$ **shows** $\forall x.\ A \longrightarrow P\ x$

[1] In the following we just write Isabelle, when we mean this combination.

proof (*intro allI impI*)
  fix x **assume** A
  with *assms* **have** $\forall\, x.\ P\ x$ **..**
  thus $P\ x$ **..**
qed

This simple example demonstrates already a lot of Isar. In general, a statement that we want to prove, consists of some (possibly empty) assumptions—that may be referenced by the name *assms* inside the proof—together with the desired conclusion. Additionally we may provide a name, here *example1*, under which to store the lemma after a successful proof (for later reference). The first step in the above proof, that is, *intro allI impI* applies the introduction rules for the universal quantifier and implication

$$\begin{array}{l} \mathit{allI}\colon\ \ (\bigwedge x.\ P\ x) \Longrightarrow \forall\, x.\ P\ x \\ \mathit{impI}\colon (P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q \end{array}$$

as often as possible, thereby transforming object-level universal quantification and implication into their meta-level counterparts. This leaves us with the meta-level goal $\bigwedge x.\ A \Longrightarrow P\ x$, that is, we have to show for an arbitrary but fixed x that under the assumption A we may conclude $P\ x$. Now, by **fix** x, we state that x is arbitrary but fixed. Then, we assume A, since we want to show an implication. Together with our global assumption $A \longrightarrow (\forall\, x.\ P\ x)$, referenced via *assms*), we can prove $\forall\, x.\ P\ x$ by some standard rule (in this case *modus ponens*; the abbreviation **..** stands for *try to solve the current goal by a single application of some standard rule*; see also Section 2.5). Again, by some standard rule (this time the specialization rule for $\forall$), we can show the desired conclusion $P\ x$. Now, the proof is complete and finished by **qed**. From this point on, we may use the name *example1* to refer to the lemma

$$\mathit{example1}\colon\ A \longrightarrow (\forall\, x.\ P\ x) \Longrightarrow \forall\, x.\ A \longrightarrow P\ x$$

which is implicitly universally quantified over A and P at the meta-level. Let us compare this proof to a proof of the same statement in apply-style.

lemma $A \longrightarrow (\forall\, x.\ P\ x) \Longrightarrow \forall\, x.\ A \longrightarrow P\ x$
apply (*intro allI impI*)
apply (*erule impE*)
apply (*assumption*)
apply (*erule allE*)

apply (*assumption*)
done

In principle those two proofs are equivalent, however, we believe that the former proof is much more readable. Hence, from now on, we ban **apply** from our repertoire of proof commands and stick to Isar proofs. Until now, we have been concerned with the *Logic* part of the above equation. So let us go for *Functional Programming*.

2.2.2 Functional Programming

Most functional programming languages provide strict typing, recursive functions and algebraic data types. This introduces a slight complication: We need to be sure that all functions that we define are total, since otherwise we could use a function definition like $f\ x = f\ x + 1$ to render our logic inconsistent (just remove $f\ x$ from both sides of the equation to obtain $0 = 1$). Hence, Isabelle is rather conservative (it has to) and only allows recursive functions where termination has been shown. Gladly, such termination proofs can often be done automatically by the system.

Many popular functions from functional programming are already part of Isabelle's library. Additionally, pervasive algebraic data types like $\alpha\ list$ or $\alpha\ option$ together with corresponding functions are available. Non-recursive functions are introduced using **definition**, whereas **fun** is used for recursive functions. Consider the following example:

definition $sqr :: nat \Rightarrow nat$ **where** $sqr\ x \equiv x * x$

fun $sqrsum :: nat\ list \Rightarrow nat$ **where**
$sqrsum\ [] = 0\ |$
$sqrsum\ (x\ \#\ xs) = sqr\ x + sqrsum\ xs$

This defines a non-recursive function *sqr* that takes the square of its argument and the recursive function *sqrsum* that takes a list of natural numbers and computes the sum of the squares of the list elements. The definition of *sqrsum* is only accepted since the system could automatically prove termination.

Isabelle already provides the functions *listsum* and *map*, so let's prove that we

could use a combination of those, instead of *sqrsum*.

lemma *sqrsum xs* = *listsum* (*map sqr xs*)
by (*induct xs*) (*simp_all add*: *sqr_def*)

The proof is amazingly short, since it can be handled automatically after telling the system that it should use induction over *xs* (every algebraic data type comes with an appropriate induction scheme) and unfold the definition of *sqr* (by default a definition of a constant *c* is stored under the name *c_def*).

2.3 Common Data Types and Functions

In this section, we give a short overview of types and (related) functions that are ubiquitous in HOL. (We introduce types in the same order as they are defined in Isabelle/HOL.)

2.3.1 Tuples

In Isabelle, n-tuples are encoded by (right-associative) nesting of pairs. Further, there is the special case of the empty tuple, represented by the type *unit* having the sole element (). The type of pairs where the first component is of type α and the second component of type β is written $\alpha \times \beta$ in Isabelle.

Reasoning about Tuples.
Pairs may be used, as if defined by the data type

datatype $\alpha \times \beta$ = *Pair* α β

We say "as if," since the internal definition is more involved. The ASCII version of $\alpha \times \beta$ is $\alpha * \beta$. As syntactic sugar, pairs may be constructed by (x, y) instead of *Pair x y*. More about tuples, can be found in the theory *Product_Type* of Isabelle/HOL.

By far the most common functions on pairs are the component-selectors:

fst $(x, y) = x$
snd $(x, y) = y$

2.3.2 Natural Numbers

For natural numbers we have the type *nat*. A natural number is either zero (represented by *0*) or the successor of another natural number n (represented by *Suc n*).

Reasoning about Natural Numbers.
Natural numbers may be used, as if defined by the data type:

datatype $nat = 0 \mid Suc\ nat$

This entails the induction scheme

$$nat.induct\colon [\![P\ 0;\ \bigwedge n.\ P\ n \Longrightarrow P\ (Suc\ n)]\!] \Longrightarrow P\ m$$

as well as the case distinction rule

$$nat.exhaust\colon [\![n = 0 \Longrightarrow P;\ \bigwedge m.\ n = Suc\ m \Longrightarrow P]\!] \Longrightarrow P$$

for natural numbers. Note that for readability, we often write $n + 1$ instead of *Suc n*. Natural numbers are defined in the theory *Nat* of Isabelle/HOL.

2.3.3 Options

The data type α *option* encapsulates optional values of type α, that is, a value having this type either contains a value x of type α (represented by *Some x*) or it is empty (represented by the constructor *None*).

Reasoning about the Option Type.
The internal definition is

datatype $\alpha\ option = None \mid Some\ \alpha$

The option type is defined in the theory *Option* of Isabelle/HOL.

One function that is particularly convenient in combination with this data type is *the*, which is defined by the equation

$$the\ (Some\ x) = x$$

2.3.4 Lists

A list containing elements of type α is either empty (represented by $[]$) or consists of some element x—the "head" of the list—followed by a shorter list xs—the "tail" of the list (represented by $x \# xs$).

Reasoning about Lists.
Lists are represented by the data type

datatype α *list* = [] | *op* # α (α *list*)

Here *op x* indicates that x is an infix operator. The ASCII variants of [] and *op* # are *Nil* and *Cons*, respectively. Again we have an induction scheme

list.induct: $[\![P\ []; \bigwedge y\ ys.\ P\ ys \Longrightarrow P\ (y \# ys)]\!] \Longrightarrow P\ xs$

and a case distinction rule

list.exhaust: $[\![xs = [] \Longrightarrow P; \bigwedge y\ ys.\ xs = y \# ys \Longrightarrow P]\!] \Longrightarrow P$

This and much more (that is, functions on lists and lemmas about those functions) can be found in the theory *List* of Isabelle/HOL.

Lists are used heavily inside IsaFoR. Hence, there are several functions on lists that pop-up regularly. Those are described in the following.

Number of Elements The number of elements contained in a list is computed by the function *length*. For brevity, we write $|xs|$ instead of *length xs*.

$|[]| = 0$
$|x \# xs| = (|xs| + 1)$

Concatenating Lists Concatenating two lists is done by the function *append* (using the infix *op* @ as syntactic sugar):

$[] @ ys = ys$
$(x \# xs) @ ys = x \# xs @ ys$

Here, it is important to note that *op* # and *op* @ are both right-associative and have the same priority. Thus, the list $1 \# (2 \# (3 \# []))$ may be written as $1 \# 2 \# 3 \# []$ and a construct like $xs @ y \# ys$ is the same as $xs @ (y \# ys)$.

Getting an Element Getting the i-th element of a list is another common task. This is done by the function *nth* having the infix syntax *op* !, such that xs ! i denotes taking the i-th element of the list xs. For brevity, we use the notation xs_i in the following. The function *nth* is defined by the equation:

$$(x \# xs)_n = (\mathit{case}\ n\ \mathit{of}\ 0 \Rightarrow x \mid k+1 \Rightarrow xs_k)$$

Taking Elements Taking the first n elements of a list is the purpose of the function *take*, defined by the equations

$$\begin{aligned} &\mathit{take}\ n\ [] = [] \\ &\mathit{take}\ n\ (x \# xs) = (\mathit{case}\ n\ \mathit{of}\ 0 \Rightarrow [] \mid m+1 \Rightarrow x \# \mathit{take}\ m\ xs) \end{aligned}$$

Dropping Elements The 'symmetric' operation to *take* is *drop*, which is defined by

$$\begin{aligned} &\mathit{drop}\ n\ [] = [] \\ &\mathit{drop}\ n\ (x \# xs) = (\mathit{case}\ n\ \mathit{of}\ 0 \Rightarrow x \# xs \mid m+1 \Rightarrow \mathit{drop}\ m\ xs) \end{aligned}$$

and whose purpose is to remove the first n elements from a list.

Mapping a Function over a List Often, we want to apply the same function to every element of a list, thereby producing a new list. This is done by *map*:

$$\begin{aligned} &\mathit{map}\ f\ [] = [] \\ &\mathit{map}\ f\ (x \# xs) = f\ x \# \mathit{map}\ f\ xs \end{aligned}$$

Converting Lists to Sets For formulating logical properties, it is mostly more convenient to reason about finite sets (since the order of elements is irrelevant) instead of lists. Therefore, the function *set* is provided, which takes a list as input and returns a set containing its elements. The definition is:

$$\begin{aligned} &\mathit{set}\ [] = \varnothing \\ &\mathit{set}\ (x \# xs) = \{x\} \cup \mathit{set}\ xs \end{aligned}$$

Since it is usually clear from the context, when a list is used as a set, we installed a syntax translation, dropping this conversion function. This results, for example, in $x \in xs$ instead of $x \in \mathit{set}\ xs$.

2.4 Combining and Modifying Facts

Very often, a fact that is needed at a certain point in a proof, can be obtained by combining several existing facts. Another time, the needed fact could be obtained by a small modification of an existing fact. In this section, we shortly explain some *attributes* that are used to achieve such tasks. Here, an *attribute* is an "argument" that may be given to a theorem.

Discharging Assumptions Consider a situation where we want to prove a fact B. Additionally, we have the facts

$AimpB$: $A \Longrightarrow B$
A: A

Since one of our facts directly satisfies the assumption of the other fact, we would like to obtain B in a neat way. This can be done using the attribute *OF* as follows:

$AimpB[OF\ A]$

In general, *OF* allows to discharge assumptions from left to right. For convenience, we may use underscores to omit positions. For example, the expression *lemma1*[*OF* _ *lemma2*] may be used to discharge the second assumption of the lemma *lemma1*, using the lemma *lemma2*.

Replacing Equals by Equals In other cases we have some fact, that is almost what we need, except that we want to apply some equations first. This can be done using the attribute *unfolded*. Consider that we have

Suc: $m = (n + 1)$
lem: $P\ m$

Then, we can directly obtain $P\ (n + 1)$ by *lem*[*unfolded Suc*].

Instantiation Remember that every lemma that has been proven in Isabelle, is implicitly universally quantified (at the meta-level) over all free variables. Sometimes it is convenient to explicitly instantiate such 'universal' lemmas.

Consider that we have proven that some property P does hold for every natural number n:

Plemma: $P\ n$

Hence, it should be possible to conclude $P\ 0$. This can easily be done by *Plemma*[*of 0*] (where we assume that P is some defined constant and hence not free in *Plemma*). Note that (like for *OF*) we may omit positions using underscores.

2.5 Some Isar Idioms

In this section we introduce some idioms that are commonly used in Isar and make life easier (and often proofs shorter).

Abbreviations Some proof methods are so common that abbreviations are provided for them. For example, every logical constant usually comes with some introduction and elimination rules. Then it is syntactically clear that whenever we try to prove a statement introducing a constant we have to use the appropriate introduction rule. Similarly, whenever we want to eliminate a constant, we use the appropriate elimination rule. Further, the system automatically distinguishes between introducing and eliminating, by just checking whether there are any current facts. If there are, then the default behavior is to eliminate the outermost constant of the first fact, otherwise, the default behavior is to introduce the outermost constant of the goal. This is done, whenever we start a proof using **proof** (but may be suppressed by using **proof** - instead), or use the method **..** (two dots) to prove something. Consider for example logical conjunction.

lemma $A \wedge B$
proof

Since we did not give a hyphen as argument to **proof**, the introduction rule *conjI* is applied, resulting in the goals:

1. A

2. B

If, on the other hand, there are current facts, the corresponding elimination rule is used. For example, in

lemma
  assumes $A \wedge B$ **shows** B
using *assms* **proof**

we have the current fact $A \wedge B$ where the outermost constant is $op \wedge$. Hence, *conjE* is applied, resulting in the goal:

1. $[\![A;\ B]\!] \Longrightarrow B$

Reasoning using Default Rules.
You may wonder, which rules exactly are considered by **proof** and **..** as default introduction and elimination rules. The answer is that default rules are declared as such in the library. Additionally, IsaFoR introduces many more default rules for newly introduced constants. The most important of them are described in 'reasoning boxes,' directly after their introduction.

Another common pattern is, when a goal is directly provable from the current fact. In such a case, the short form **.** (a single dot), may be used. For example,

from *TrueI* **have** *True* **.**

Where *TrueI* is a lemma from Isabelle/HOL, stating that *True* is always a valid fact.

Collecting Facts Often, we have to prove the assumptions of a rule one after the other, before it is possible to apply it. It would be tedious to invent new names for every intermediate fact. Hence, it is possible to collect a bunch of facts and apply them simultaneously. The general scheme is:

have A_1 ⟨*proof omitted*⟩
moreover have A_2 ⟨*proof omitted*⟩

⋮

moreover have A_n ⟨*proof omitted*⟩

ultimately have B **by** (*rule* R)

where R is a rule that yields B and whose assumptions are satisfied by A_1 to A_n (in this order).

Obtaining Witnesses Usually, when we eliminate an existential quantifier from a term like $\exists\, x.\ P\ x$, we want to get hold of a concrete witness satisfying the property P. In Isar, this is done using **obtain**, for specifying the name(s) of the witness(es) and **where**, for specifying the properties that they have to satisfy. The simplest example would be:

have $\exists\, x.\ P\ x$ ⟨*proof omitted*⟩
then obtain y **where** $P\ y$ **..**

2.6 Chapter Notes

In this chapter, we gave a short introduction to interactive theorem proving, which is part of what is called *mechanized proof*. MacKenzie [26] gives a nice historical overview on mechanized proof and discusses some related philosophical questions.

Afterwards, we gave an overview of Isabelle's meta-logic, which is described by Paulson [31]. Since our discussion of Pure was only brief, we refer to the thorough documentation provided by Wenzel [49, 50] as further reading. There is also a must-see tutorial on Isabelle/HOL by Nipkow et al. [29].

Then, we shortly discussed how logic and functional programming is part of HOL. By Isar, Isabelle provides a human readable style of proof. Isar was introduced by Wenzel [48]. For a stepwise introduction to Isar we refer to Nipkow [28]. Concerning functional programming, we have to make sure that every newly introduced function is terminating. Gladly, most of the time a user does not have to do termination proofs for newly introduced functions by hand, thanks to the ingenious definitional function package by Krauss [25].

Finally, we listed some data types and functions that are used throughout our formalizations; showed a neat way to combine and modify facts; and indicated some structures that can be found in many Isar style proofs.

Chapter 3

Abstract Rewriting

An abstraction is one thing
that represents several real things equally well.

Edsger Wybe Dijkstra

Abstract rewriting is rewriting on abstract entities, whereby we mean that we do not have any information on the kind of entities. Consequently, every fact that we are able to prove for abstract rewriting, does also hold for any specific structure of entities. Hence, we may freely use the results of this chapter for term rewriting (from Chapter 4 on). In general, rewriting is the process of applying rules from a given set to some entity that is being rewritten. This is of course only possible if there is a rule for the specific entity under consideration. Additionally, more than one rule may be applicable. This is usually modeled using an *abstract rewrite system* (ARS).

Definition 3.1 (Abstract Rewrite Systems). An ARS $\mathcal{A}$ is a pair $(A, \to_{\mathcal{A}})$, consisting of a *carrier* A, together with a binary relation $\to_{\mathcal{A}} \in A \times A$.

Example 3.1. Consider the carrier $A = \{a, b, c\}$ together with the binary relation $\to = \{(a, a), (a, b), (a, c)\} \subseteq A \times A$. We have three rules for a, but none for the other two carrier elements. A single rewrite step from a to b (denoted by $a \to b$) is possible, due to the rule (a, b).

3.1 Abstract Rewriting in Isabelle

Since we are in a typed setting (using HOL), we do omit a specific carrier in our formalizations and just use the type

types $\alpha\ ars = (\alpha \times \alpha)\ set$

of *endorelations* over type α to denote ARSs in IsaFoR. (In general, binary relations may have different types for their domain and codomain, that is, a binary relation has the type $(\alpha \times \beta)\ set$. To stress that we are only considering binary relations for which the domain and codomain coincide, we use the term endorelation.) If $(a,\ b) \in \mathcal{A}$, we say that there is a *rewrite step* from a to b. Many facts about binary relations are already provided in the theory *Relation* of Isabelle's library. For example: definitions for the (*reflexive* and) *transitive closure* R^+ (R^*) of a relation R, and the *composition* of two binary relations R and S, given by

$$rel\_comp\_def\colon\ R \circ S \equiv \{(x,\ z) \mid \exists\, y.\ (x,\ y) \in R \wedge (y,\ z) \in S\}$$

When $(a,\ b) \in \mathcal{A}^*$ $(\mathcal{A}^+)$, we say that there is a (non-empty) *rewrite sequence* from a to b (or simply that *a rewrites to b*).

Reasoning about Binary Relations.
Many facts about binary relations can be found in the theory *Relation*, which is part of Isabelle/HOL. Just note that the ASCII versions of composition and the (reflexive and) transitive closure are: O (the capital letter 'O') and $\hat{}+$ ($\hat{}*$). Consult *Relation*, for further details.

Definition 3.2 (Termination / Strong Normalization). It is sometimes convenient to analyze the termination behavior of single elements. We say that an element a is terminating (with respect to an ARS $\mathcal{A}$), whenever there is no infinite $\mathcal{A}$-sequence starting at a:

$$\mathsf{SN}_{\mathcal{A}}(a) \equiv \nexists\, S.\ S\ 0 = a \wedge (\forall\, i.\ (S\ i,\ S\ (i+1)) \in \mathcal{A})$$

An infinite sequence of elements of type α is modeled by a function $S{::}nat \Rightarrow \alpha$. Further, by demanding that between every element and its successor there is an $\mathcal{A}$-step, the infinite sequence is guaranteed to be an infinite rewrite sequence of $\mathcal{A}$.

Now, an ARS $\mathcal{A}$ is *terminating*, if every carrier element is terminating:

$$\mathsf{SN}(\mathcal{A}) \equiv \forall\, a.\ \mathsf{SN}_{\mathcal{A}}(a)$$

Reasoning about Termination.
The ASCII version of the strong normalization predicate (for single elements) is *SN* (*SN_elt*). Note that strong normalization is connected to the Isabelle internal notion of *well-foundedness* by the lemma

SN_iff_wf: $\mathsf{SN}(\mathcal{A}) = wf\ (\mathcal{A}^{-1})$

where $\mathcal{A}^{-1}$ denotes the inverse of the relation $\mathcal{A}$. The ASCII variant is *^-1*.

The above two definitions for strong normalization are available in IsaFoR under the name *SN_defs*. There are also some introduction and elimination rules for strong normalization. For $\mathsf{SN}_{\mathcal{A}}(a)$, the most important rules are

SN_elt_I: $(\bigwedge S.\ [\![S\ 0 = a;\ \forall i.\ (S\ i,\ S\ (i+1)) \in \mathcal{A}]\!] \Longrightarrow False) \Longrightarrow \mathsf{SN}_{\mathcal{A}}(a)$
SN_elt_E: $[\![\mathsf{SN}_{\mathcal{A}}(a);\ \nexists S.\ S\ 0 = a \wedge (\forall i.\ (S\ i,\ S\ (i+1)) \in \mathcal{A}) \Longrightarrow P]\!] \Longrightarrow P$

and for $\mathsf{SN}(\mathcal{A})$ we have

SN_I: $(\bigwedge a.\ \mathsf{SN}_{\mathcal{A}}(a)) \Longrightarrow \mathsf{SN}(\mathcal{A})$
SN_I': $(\bigwedge S.\ \forall i.\ (S\ i,\ S\ (i+1)) \in \mathcal{A} \Longrightarrow False) \Longrightarrow \mathsf{SN}(\mathcal{A})$
SN_E: $[\![\mathsf{SN}(\mathcal{A});\ \mathsf{SN}_{\mathcal{A}}(a) \Longrightarrow P]\!] \Longrightarrow P$
SN_E': $[\![\mathsf{SN}(\mathcal{A});\ \nexists S.\ \forall i.\ (S\ i,\ S\ (i+1)) \in \mathcal{A} \Longrightarrow P]\!] \Longrightarrow P$

Further, the principle of well-founded induction that is already present in Isabelle, carries over to strong normalization:

SN_induct: $[\![\mathsf{SN}(A);\ \bigwedge a.\ (\bigwedge b.\ (a,\ b) \in A \Longrightarrow P\ b) \Longrightarrow P\ a]\!] \Longrightarrow P\ a$

When conceiving a rewrite sequence of an ARS as a computation sequence of a program, where the involved elements denote the intermediate states of the program, termination of the ARS is equivalent to termination of the program. That is, we have an answer to the question: *does every computation path lead to a result eventually?* There are other interesting questions concerning programs. One of them is: *Is the order in which we evaluate expressions irrelevant, that is, is it always possible to join two different computation paths?* On the abstract rewriting level, this second question is known as *confluence* (or the *Church-Rosser property*). Like termination, confluence is undecidable in general. However, there is a nice connection between termination and confluence: *Newman's Lemma*. To get further acquainted with Isabelle, in the next section, we will give a proof of *Newman's Lemma* as it is part of IsaFoR. Readers already familiar with Isabelle, may safely skip the next section, since confluence of ARSs is not needed in our later formalizations.

3.2 Newman's Lemma

Our main goal, is to imitate the textbook proof of Newman's Lemma as closely as possible.

Definition 3.3 (Confluence / Church-Rosser). Before we can actually state the lemma, we need to define (*local*) *confluence* (also known as the (*Weak*) *Church-Rosser Property*; explaining the following names). For single elements, this is done as follows:

$$\mathsf{WCR}_{\mathcal{A}}(a) \equiv \forall\, b\ c.\ (a,\ b) \in \mathcal{A} \wedge (a,\ c) \in \mathcal{A} \longrightarrow (b,\ c) \in \mathcal{A}^{\downarrow}$$
$$\mathsf{CR}_{\mathcal{A}}(a) \equiv \forall\, b\ c.\ (a,\ b) \in \mathcal{A}^{*} \wedge (a,\ c) \in \mathcal{A}^{*} \longrightarrow (b,\ c) \in \mathcal{A}^{\downarrow}$$

where $\mathcal{A}^{\downarrow}$ is the *joinability relation* of $\mathcal{A}$, given by $\mathcal{A}^{\downarrow} \equiv \mathcal{A}^{*} \circ (\mathcal{A}^{-1})^{*}$. For an ARS $\mathcal{A}$ we say that it is (locally) confluent, whenever every element is (locally) confluent.

Reasoning about Confluence.
The corresponding ASCII constants for the (Weak) Church-Rosser Property and the joinability relation are (*WCR_elt*/*WCR*) *CR_elt*/*CR* and *join*. Again, there are the obvious introduction and elimination rules:

WCR_elt_I: $(\bigwedge b\ c.\ [\![(a,\ b) \in \mathcal{A};\ (a,\ c) \in \mathcal{A}]\!] \Longrightarrow (b,\ c) \in \mathcal{A}^{\downarrow}) \Longrightarrow \mathsf{WCR}_{\mathcal{A}}(a)$
WCR_elt_E:
$[\![\mathsf{WCR}_{\mathcal{A}}(a);\ (b,\ c) \in \mathcal{A}^{\downarrow} \Longrightarrow P;\ (a,\ b) \notin \mathcal{A} \Longrightarrow P;\ (a,\ c) \notin \mathcal{A} \Longrightarrow P]\!] \Longrightarrow P$
CR_elt_I: $(\bigwedge b\ c.\ [\![(a,\ b) \in \mathcal{A}^{*};\ (a,\ c) \in \mathcal{A}^{*}]\!] \Longrightarrow (b,\ c) \in \mathcal{A}^{\downarrow}) \Longrightarrow \mathsf{CR}_{\mathcal{A}}(a)$
CR_elt_E:
$[\![\mathsf{CR}_{\mathcal{A}}(a);\ (b,\ c) \in \mathcal{A}^{\downarrow} \Longrightarrow P;\ (a,\ b) \notin \mathcal{A}^{*} \Longrightarrow P;\ (a,\ c) \notin \mathcal{A}^{*} \Longrightarrow P]\!] \Longrightarrow P$
WCR_I: $(\bigwedge a.\ \mathsf{WCR}_{\mathcal{A}}(a)) \Longrightarrow \mathsf{WCR}(\mathcal{A})$
WCR_E: $[\![\mathsf{WCR}(\mathcal{A});\ \mathsf{WCR}_{\mathcal{A}}(a) \Longrightarrow P]\!] \Longrightarrow P$
CR_I: $(\bigwedge a.\ \mathsf{CR}_{\mathcal{A}}(a)) \Longrightarrow \mathsf{CR}(\mathcal{A})$
CR_E: $[\![\mathsf{CR}(\mathcal{A});\ \mathsf{CR}_{\mathcal{A}}(a) \Longrightarrow P]\!] \Longrightarrow P$

The only difference between local confluence and confluence, is that the former merely demands that two single steps are joinable, whereas the latter demands that every two rewrite sequences starting at the same element are joinable.

Lemma 3.1 (Newman's Lemma). *An ARS $\mathcal{A}$ is confluent, whenever it is terminating and locally confluent. In Isabelle this is stated as follows:*

$$[\![\mathsf{SN}(\mathcal{A});\ \mathsf{WCR}(\mathcal{A})]\!] \Longrightarrow \mathsf{CR}(\mathcal{A})$$

In the remainder of this section, we give a proof of the above statement.

lemma *Newman*: **assumes** $\mathsf{SN}(\mathcal{A})$ **and** $\mathsf{WCR}(\mathcal{A})$ **shows** $\mathsf{CR}(\mathcal{A})$
proof

We have to show that an arbitrary but fixed element x is confluent. We do this using the first assumption and proceeding by well-founded induction.

fix x
from ‹$\mathsf{SN}(\mathcal{A})$› **show** $\mathsf{CR}_{\mathcal{A}}(x)$
proof *induct*
case (*IH* x) **show** $\mathsf{CR}_{\mathcal{A}}(x)$

At this point, the induction hypothesis *IH*: $(x, b) \in \mathcal{A} \Longrightarrow \mathsf{CR}_{\mathcal{A}}(b)$ for arbitrary b is at our disposal.

proof
fix y z

In order to show confluence for the single element x, we assume that there are two rewrite sequences to the arbitrary but fixed elements y and z and then have to show that those are joinable.

assume $(x, y) \in \mathcal{A}^*$ **and** $(x, z) \in \mathcal{A}^*$
from ‹$(x, y) \in \mathcal{A}^*$› **obtain** m **where** $(x, y) \in \mathcal{A}^m$ **..**
from ‹$(x, z) \in \mathcal{A}^*$› **obtain** n **where** $(x, z) \in \mathcal{A}^n$ **..**

We obtain the lengths m and n of the two rewrite sequences from x to y and from x to z. The remainder of the proof runs by nested case distinctions on m and n. Since the cases where either of them is *0* are trivial, we omit them and just present the interesting case, that is, $m = m' + 1$ and $n = n' + 1$. To get the proof idea, Figure 3.1 may be helpful.

show $(y, z) \in \mathcal{A}^{\downarrow}$
proof (*cases* n)
case *0* **show** *?thesis* ⟨*proof omitted*⟩
next
case (*Suc* n')
from ‹$(x, z) \in \mathcal{A}^n$›[*unfolded Suc*] **obtain** t
where $(x, t) \in \mathcal{A}$ **and** $(t, z) \in \mathcal{A}^*$ **..**

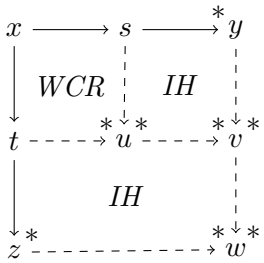

Figure 3.1: Proof sketch of *Newman's Lemma*.

show *?thesis*
proof (*cases m*)
  case *0* **show** *?thesis* ⟨*proof omitted*⟩
next
  case (*Suc m'*)
  from ‹$(x, y) \in \mathcal{A}^m$›[*unfolded Suc*] **obtain** *s*
    where $(x, s) \in \mathcal{A}$ **and** $(s, y) \in \mathcal{A}^*$ **..**

First we complete the top left square of Figure 3.1, using the assumption that $\mathcal{A}$ is locally confluent. In this way we obtain the element u.

from ‹*WCR* $\mathcal{A}$› **and** ‹$(x, s) \in \mathcal{A}$› **and** ‹$(x, t) \in \mathcal{A}$›
  have $(s, t) \in \mathcal{A}^\downarrow$ **by** *auto*
then obtain *u* **where** $(s, u) \in \mathcal{A}^*$ **and** $(t, u) \in \mathcal{A}^*$ **..**

Then, using the induction hypothesis instantiated with $(x, s) \in \mathcal{A}$ (that is, we discharge the first assumption of *IH* by *OF* ...), we complete the top right square. Obtaining the element v. (Recall that **.** stands for a proof by assumption.)

from *IH*[*OF* ‹$(x, s) \in \mathcal{A}$›] **have** $\mathsf{CR}_{\mathcal{A}}(s)$ **.**
with ‹$(s, u) \in \mathcal{A}^*$› **and** ‹$(s, y) \in \mathcal{A}^*$› **have** $(u, y) \in \mathcal{A}^\downarrow$ **by** *auto*
then obtain *v* **where** $(u, v) \in \mathcal{A}^*$ **and** $(y, v) \in \mathcal{A}^*$ **..**

Finally, we complete the bottom square, using the induction hypothesis instantiated with $(x, t) \in \mathcal{A}$. (Note that the abbreviation *?thesis* may always be used to refer to the conclusion of the current subproof.)

```
      from IH[OF ⟨(x, t) ∈ A⟩] have CR_A(t) .
      moreover from ⟨(t, u) ∈ A*⟩ and ⟨(u, v) ∈ A*⟩
        have (t, v) ∈ A* by auto
      ultimately have (z, v) ∈ A↓ using ⟨(t, z) ∈ A*⟩ by auto
      then obtain w where (z, w) ∈ A* and (v, w) ∈ A* ..
      from ⟨(y, v) ∈ A*⟩ and ⟨(v, w) ∈ A*⟩ have (y, w) ∈ A* by auto
      with ⟨(z, w) ∈ A*⟩ show ?thesis by auto
    qed
  qed
 qed
qed
```

In the next section we will give a proof of a fact that is used several times in IsaFoR.

3.3 Non-Strict Ending

When reasoning about termination, we are often concerned with infinite sequences (as we have already seen above, those are modeled by functions from the natural numbers to the type of elements the sequence has). This is mostly used in *proofs by contradiction*, where we show that the existence of an infinite sequence would lead to a contradiction.

In the following, we give a particularly useful lemma, which is used several times in IsaFoR.

Lemma 3.2. *Consider two ARSs $\mathcal{A}$ and $\mathcal{B}$, as well as the infinite $\mathcal{A} \cup \mathcal{B}$-sequence S. Let the first element of S be strongly normalizing with respect to $\mathcal{B}$, and $\mathcal{A}$ be* compatible *to $\mathcal{B}$ (that is, an $\mathcal{A}$-step followed by a $\mathcal{B}$-step, can be replaced by a single $\mathcal{B}$-step). Then, after a finite number of steps, the remaining sequence consists solely of $\mathcal{A}$-steps.*

In usual applications of this lemma $\mathcal{A}$ is a non-strict relation, hence the name.

lemma *non_strict_ending*:
  assumes *seq*: $\forall\, i.\ (S\ i,\ S\ (i+1)) \in \mathcal{A} \cup \mathcal{B}$

and $\mathcal{A} \circ \mathcal{B} \subseteq \mathcal{B}$ **and** $\mathsf{SN}_{\mathcal{B}}(S\ 0)$
shows $\exists j.\ \forall i{\geq}j.\ (S\ i,\ S\ (i+1)) \in \mathcal{A} - \mathcal{B}$

For the sake of a contradiction, we assume that the conclusion does not hold.

proof (*rule ccontr*)
assume $\neg\ (\exists j.\ \forall i{\geq}j.\ (S\ i,\ S\ (i+1)) \in \mathcal{A} - \mathcal{B})$

But then, by the Axiom of Choice, we get hold of a selection function π over the indices, such that there is a $\mathcal{B}$-step from position $\pi\ i$ to position $\pi\ (i+1)$ and additionally π is monotone (that is, $\forall i.\ i \leq \pi\ i$).

with *seq* **have** $\forall j.\ \exists i.\ i \geq j \wedge (S\ i,\ S\ (i+1)) \in \mathcal{B}$ **by** *blast*
from *choice*[*OF this*] **obtain** π
where *B_steps*: $\forall i.\ \ \pi\ i \geq i \wedge (S\ (\pi\ i),\ S\ ((\pi\ i)+1)) \in \mathcal{B}$ **..**

The idea is now to use π, in order to pick those elements from S that are connected by $\mathcal{B}^+$. To this end, we need an auxiliary function that actually constructs indices for the new sequence from indices of the old one. This is the purpose of

shift_by f 0 $=$ *0*
shift_by f $(i+1) = ((f\ (shift\_by\ f\ i)) + 1)$

With the help of this function, we define an infinite sequence *?S* that pinpoints those elements of the original sequence S which are connected by $\mathcal{B}^+$.

let *?S i* $=$ $S\ (shift\_by\ \pi\ i)$

Before we are actually able to show that the new sequence *?S* is an infinite sequence of $\mathcal{B}^+$-steps, we show the following:

have $\forall i.\ (S\ i,\ S\ ((\pi\ i)+1)) \in \mathcal{B}^+$

For this proof we need two more lemmas: One that shows that in an infinite $\mathcal{B}$-sequence there is a step from i to j in the reflexive and transitive closure of $\mathcal{B}$, whenever $i \leq j$. And another one that shows that if a relation $\mathcal{A}$ is *compatible* with a relation $\mathcal{B}$, that is $\mathcal{A} \circ \mathcal{B} \subseteq \mathcal{B}$, then an arbitrary $\mathcal{A} \cup \mathcal{B}$-sequence, followed by a $\mathcal{B}$-step, may be combined into a non-empty $\mathcal{B}$ sequence. In Isabelle those two lemmas look as follows:

iseq_imp_steps:
$[\![\forall i.\ (S\ i,\ S\ (i+1)) \in \mathcal{A};\ i \leq j]\!] \Longrightarrow (S\ i,\ S\ j) \in \mathcal{A}^*$
comp_rtrancl_trancl:
$[\![\mathcal{A} \circ \mathcal{B} \subseteq \mathcal{B};\ (x,\ y) \in (\mathcal{A} \cup \mathcal{B})^* \circ \mathcal{B}]\!] \Longrightarrow (x,\ y) \in \mathcal{B}^+$

proof
  fix *i*
  from *B_steps* **have** $\pi\ i \geq i$
    and *B_step*: $(S\ (\pi\ i),\ S\ ((\pi\ i)+1)) \in \mathcal{B}$ **by** *auto*
  from *seq* **and** ‹$\pi\ i \geq i$›
    have $(S\ i,\ S\ (\pi\ i)) \in (\mathcal{A} \cup \mathcal{B})^*$ **by** (*rule iseq_imp_steps*)
  with *B_step* **have** $(S\ i,\ S\ ((\pi\ i)+1)) \in (\mathcal{A} \cup \mathcal{B})^* \circ \mathcal{B}$ **by** *auto*
  with ‹$\mathcal{A} \circ \mathcal{B} \subseteq \mathcal{B}$› **show** $(S\ i,\ S\ ((\pi\ i)+1)) \in \mathcal{B}^+$
    by (*rule comp_rtrancl_trancl*)
qed
hence $\forall i.\ (?S\ i,\ ?S\ (i+1)) \in \mathcal{B}^+$ **by** *simp*

It remains to show that this contradicts the assumption $\mathsf{SN}_{\mathcal{B}}(S\ 0)$. Therefore, we need a lemma, stating that the termination behavior of a single element under $\mathcal{B}$, carries forward to $\mathcal{B}^+$, that is:

SN_elt_imp_SN_elt_trancl: $\mathsf{SN}_{\mathcal{A}}(x) \Longrightarrow \mathsf{SN}_{\mathcal{A}^+}(x)$

moreover from ‹$\mathsf{SN}_{\mathcal{B}}(S\ 0)$› **have** $\mathsf{SN}_{\mathcal{B}^+}(?S\ 0)$
  using *SN_elt_imp_SN_elt_trancl* **by** *simp*
ultimately show *False* **unfolding** *SN_defs* **by** *best*
qed

Now we know everything about termination of ARSs that we need in the upcoming chapters. In the next chapter we will give structure to the elements of our rewrite systems. That is, we are concerned with the rewriting of terms.

3.4 Chapter Notes

In this chapter, we started by introducing our Isabelle formalization of abstract rewriting [36]. As an example of its usage, we gave a proof of Newman's Lemma. Note that this is by far not the first formalization of Newman's Lemma. There

are formalizations in ACL2 [33], Boyer-Moore [34], Coq [19], Isabelle [32], Otter [3], and PVS [11], among others. Yet another formalization is part of the Isabelle sources since 1995 (in the theory *Library/Commutation*). It tries to exhaust automatic methods—giving incremental versions of the lemma that are using more and more automation—resulting in an impressively short proof.

Chapter 4

Term Rewriting

Don't ask for its meaning ask for its rules!

Vincent van Oostrom

Abstract rewriting is convenient for establishing properties and proving facts that do not rely on the structure of elements being rewritten. However, it is not that useful when we want to prove termination of a program. The problem is to fix an adequate abstraction. If we do not abstract at all, writing down the ARS corresponding to a program, is only possible when we already know its termination behavior. It is also possible to loose termination by the abstraction.

Example 4.1. Consider the following Haskell implementation of the *Ackermann function*:

```
ack 0 y = y + 1                          (1)
ack (x + 1) 0 = ack x 1                  (2)
ack (x + 1) (y + 1) = ack x (ack (x + 1) y)   (3)
```

Now, let us represent this program as an ARS. One way would be to abstract from concrete integers and just consider function calls. This would result in the graph depicted in Figure 4.1. But, as is evident from the cycles in the graph, this ARS is not terminating, even though the original program is.

Another possibility would be to identify a function call with its result. Then the carrier would consist of all the integers (restricted to a fixed number of bits) that are results of some call to `ack`, but there would be no edges (that is, we are considering an empty binary relation). This abstraction trivially establishes

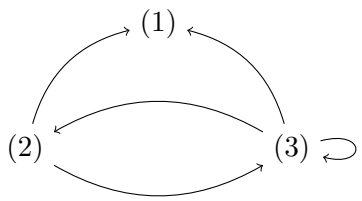

Figure 4.1: The call-graph of the *Ackermann function*.

termination of the *Ackermann* program. However, we had to exploit the fact that every call to `ack` yields a result in the first place. Hence, this abstraction is also not useful for termination analysis, since termination is already required to build it.

What we would really like to have is some structure on elements that can be used to represent facts like changes in function arguments, the nesting structure of functions, mutual dependencies, etc. There are several possibilities: strings, first-order terms, lambda terms, graphs, and what is more. For our purposes we choose first-order terms, which give a good compromise between expressiveness and simplicity.

4.1 First-Order Terms

In the literature *terms* are usually built over some *signature* of *function symbols* having a certain *arity*, together with a (usually countable infinite) set of *variables*. Then a term is either a variable, or a function application with a number of arguments, corresponding to the arity of the involved function symbol. For example, the left-hand sides of the *Ackermann* program from Example 4.1, could be written as $\mathsf{ack}(\mathsf{0}, y)$, $\mathsf{ack}(\mathsf{s}(x), \mathsf{0})$ and $\mathsf{ack}(\mathsf{s}(x), \mathsf{s}(y))$, respectively (where s denotes the successor function).

As already for the carrier in the abstract rewriting setting, we choose to represent variables and function symbols by types instead of sets (which is somehow equivalent anyway). In IsaFoR we use the following data type:

datatype (α, β) *term* = *Var* β | *Fun* α ((α, β) *term list*)

This denotes terms built over function symbols of type α and variables of type β. Note that there is no well-formedness restriction to function applications in the sense that some arity has to be respected. This also implies that the name of a function symbol alone, is not enough to identify it uniquely, since the same name could be used with different arities. Thus, we usually take pairs $(f,\ n)$ of function symbols and arities, to gain uniqueness. Using this data type, the above terms would be represented by

Fun "ack" [Fun "0" [], Var "y"]
Fun "ack" [Fun "s" [Var "x"], Fun "0" []]
Fun "ack" [Fun "s" [Var "x"], Fun "s" [Var "y"]]

where we use *string*s to represent function symbols and variables. In examples, we use the more compact notation from above (for example, s(0) instead of *Fun "s" [Fun "0" []]*), where function symbols are written sans serif, to distinguish them from variables (which are written in *italics*).

Reasoning about Terms.
As for every data type, Isabelle provides case distinction and induction for terms. Case distinction is:

term.exhaust: $[\![\bigwedge x.\ t = Var\ x \Longrightarrow P;\ \bigwedge f\ ts.\ t = Fun\ f\ ts \Longrightarrow P]\!] \Longrightarrow P$

The automatically generated induction scheme for terms, however, is a bit bulky, due to the nested recursive structure of terms and lists in the *Fun*-case (we deliberately omit its name, since it should not be used):

$[\![\bigwedge x.\ P\ (Var\ x);\ \bigwedge f\ ts.\ Q\ ts \Longrightarrow P\ (Fun\ f\ ts);\ Q\ []$;

$\bigwedge t\ ts.\ [\![P\ t;\ Q\ ts]\!] \Longrightarrow Q\ (t\ \#\ ts)]\!]$

$\Longrightarrow P\ t \wedge Q\ ts$

For a more textbook-like induction over terms we introduce a new induction scheme (which is also installed as the default induction scheme for terms):

term_induct:
$[\![\bigwedge x.\ P\ (Var\ x);\ \bigwedge f\ ss.\ (\bigwedge s.\ s \in ss \Longrightarrow P\ s) \Longrightarrow P\ (Fun\ f\ ss)]\!] \Longrightarrow P\ t$

Definition 4.1 (Root Symbols). For a function application $t = Fun\ f\ ts$, we call f the *root symbol* (or simply *root*) of the term t. In IsaFoR, we have

root (Var x) = None
root (Fun f ts) = Some f

where the fact that the root is only defined for function applications, is expressed by using HOL's *option* type. (Note that in the literature, the root of a variable is sometimes defined to be the variable itself. In a typed setting, such a definition is not possible.) For readability, the term *the* (*root* t) is abbreviated to $\mathsf{root}(t)$, in the remainder.

Definition 4.2 (Variables occuring in a Term). The set of variables occurring in a term t is defined as follows:

$\mathcal{V}\mathsf{ar}(\mathit{Var}\ x) = \{x\}$
$\mathcal{V}\mathsf{ar}(\mathit{Fun}\ f\ ts) = (\bigcup_{t \in ts}\ \mathcal{V}\mathsf{ar}(t))$

In addition, $\mathcal{V}$ denotes the set of all *Var* terms, that is, a term t is a variable, if and only if it is a member of $\mathcal{V}$.

$t \in \mathcal{V} \equiv \exists\, x.\ t = \mathit{Var}\ x$
$t \notin \mathcal{V} \equiv \exists\, f\ ts.\ t = \mathit{Fun}\ f\ ts$

Reasoning about Variables.
The ASCII constant *vars_term* is used for the set of all variables occurring in a term. Further, there is the function *is_var*, to check whether a term is a variable. Then, membership in $\mathcal{V}$ is realized by *is_var* $t = t \in \mathcal{V}$.

Definition 4.3 (Function Symbols occuring in a Term). The set of function symbols (paired with their respective arities) occurring in a term t is given by the recursive equations:

$\mathcal{F}\mathsf{un}(\mathit{Var}\ x) = \varnothing$
$\mathcal{F}\mathsf{un}(\mathit{Fun}\ f\ ts) = \{(f,\ |ts|)\} \cup (\bigcup_{t \in ts}\ \mathcal{F}\mathsf{un}(t))$

Reasoning about Function Symbols.
Internally, the constant *funas_term* is used to collect all the function symbol arity pairs occurring in a term. The 'a' in 'funas' indicates that also the arity is considered. This means that in IsaFoR, two function symbols having the same name, but different arities, are considered to be different (as for example also in Prolog).

4.1.1 Auxiliary Functions on Terms

In this section, we give a short list of functions that are sometimes useful to write down facts about terms more succinctly. The *arguments* of a term are

defined by the equations:

$args\ (Var\ x) = []$
$args\ (Fun\ f\ ts) = ts$

Moreover, the *number of arguments* is defined by:

$num\_args\ t = |args\ t|$

4.2 Term Rewrite Systems

To describe the semantics of a program we use *term rewrite systems* (TRSs).

Definition 4.4 (Term Rewrite Systems). A TRS $\mathcal{R}$ is a set of (*rewrite*) *rules*, where each rule is a pair of terms $(l,\ r)$, having the *left-hand side* l and the *right-hand side* r.

To facilitate the reading from left to right, we sometimes write $l \to r$ instead of (l, r). In IsaFoR, we have the following types for rules and TRSs:

types
$(\alpha,\ \beta)\ rule = (\alpha,\ \beta)\ term \times (\alpha,\ \beta)\ term$
$(\alpha,\ \beta)\ trs = (\alpha,\ \beta)\ rule\ set$

Intuitively, every rule corresponds to one case of a function definition.

Definition 4.5 (Well-Formed TRSs). A TRS $\mathcal{R}$ is said to be *well-formed*, whenever all left-hand sides are non-variable terms and every variable that occurs in a right-hand side also occurs in the corresponding left-hand side.

$\mathsf{WF}(\mathcal{R}) \equiv \forall\,(l,\ r){\in}\mathcal{R}.\ l \notin \mathcal{V} \wedge \mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(l)$

Reasoning about Well-Formed TRSs.
The ASCII version of $\mathsf{WF}(\mathcal{R})$ is *wf_trs* $\mathcal{R}$. Its original (and of course equivalent) definition without syntax translations is the following:

$wf\_trs\ \mathcal{R} = (\forall\, l\ r.\ (l,\ r) \in \mathcal{R} \longrightarrow (\exists\, f\ ts.\ l = Fun\ f\ ts) \wedge vars\_term\ r \subseteq vars\_term\ l)$

Definition 4.6 (Signatures). A *signature* is a set of function symbol arity pairs. Here, the arity of a function symbol determines, to how many arguments it is

applied. Often, we are interested in the signature of symbols actually occurring in a TRS $\mathcal{R}$. This is defined as follows:

$$\mathcal{F}(\mathcal{R}) \equiv \bigcup_{(l,\, r)\in\mathcal{R}} \mathcal{F}\mathsf{un}(l) \cup \mathcal{F}\mathsf{un}(r)$$

Reasoning about Signatures.
The signature of a TRS $\mathcal{R}$ is computed by the functions *funas_rule* and *funas_term* as follows:

$$\mathit{funas\_rule}\ r \equiv \mathit{funas\_term}\ (\mathit{fst}\ r) \cup \mathit{funas\_term}\ (\mathit{snd}\ r)$$

$$\mathit{funas\_trs}\ \mathcal{R} \equiv \bigcup r{\in}\mathcal{R}.\ \mathit{funas\_rule}\ r$$

Here, *fst* and *snd* are predefined selector-functions for the first and the second component of a pair, respectively. All these definitions can be unfolded using the fact *funas_defs*.

Definition 4.7 (Defined Symbols and Constructors). The root symbols of the left-hand sides of a TRS $\mathcal{R}$, determine which functions are defined, hence, those are called the *defined symbols* of $\mathcal{R}$. The set of defined symbols of a given TRS $\mathcal{R}$ is given by

$$(f,\, n) \in \mathcal{D}_{\mathcal{R}} \equiv \exists\, l.\ (\exists\, r.\ (l,\, r) \in \mathcal{R}) \wedge \mathit{root}\ l = \mathit{Some}\ f \wedge \mathit{num\_args}\ l = n$$

Function symbols that are not defined are called *constructor symbols* (or simply *constructors*). Whereas defined symbols represent the functions that are defined by a TRS, constructors represent data that may be manipulated by those functions.

Reasoning about Defined Symbols.
Internally, the basis is the function *defined* that checks, whether a given symbol occurs as a root of some left-hand side in a given TRS $\mathcal{R}$:

$$\mathit{defined}\ \mathcal{R}\ (f,\, n) \equiv \exists\, l\ r.\ (l,\, r) \in \mathcal{R} \wedge \mathit{root}\ l = \mathit{Some}\ f \wedge \mathit{num\_args}\ l = n$$

Then the set of defined symbols is the collection:

$$\mathit{defs}\ \mathcal{R} \equiv \{(f,\, n).\ \mathit{defined}\ \mathcal{R}\ (f,\, n)\}$$

Example 4.2. The definition of the *Ackermann function* in Example 4.1, can

be encoded by the TRS

$$\begin{aligned} \mathsf{ack}(\mathsf{0}, y) &\to \mathsf{s}(y) \\ \mathsf{ack}(\mathsf{s}(x), \mathsf{0}) &\to \mathsf{ack}(x, \mathsf{s}(\mathsf{0})) \\ \mathsf{ack}(\mathsf{s}(x), \mathsf{s}(x)) &\to \mathsf{ack}(x, \mathsf{ack}(\mathsf{s}(x), y)) \end{aligned}$$

where we use a unary representation of natural numbers, constructed over *zero* (0), and the *successor function* (s). (In the following we refer to thus built numbers as *Peano numbers*.) The only defined symbol is ack.

As in function definitions of programming languages, the intention is that such rules may be applied in arbitrary contexts and that parameters (that is, variables in left-hand sides of rules) may be instantiated by arbitrary inputs. Hence, before we can formally state the semantics of a TRS, we need to define *contexts* and *substitutions*.

4.3 Contexts

Closely related to terms are *contexts*. A context is like a term, except that it contains a *hole* that acts as a placeholder for arbitrary terms that may be plugged in later. The corresponding Isabelle data type is:

datatype (α, β) *ctxt* =
  Hole |
  More α ((α, β) *term list*) ((α, β) *ctxt*) ((α, β) *term list*)

For convenience the empty context (*Hole*) may also be denoted by $\Box$. The *More* constructor is almost the same as *Fun* for terms, except that the argument list is split into three parts: the arguments to the left of the one containing the hole, the argument containing the hole, and the arguments to the right of the one containing the hole.

Definition 4.8 (Contexts around Terms). Plugging a term inside a context is defined by

$\Box[t] = t$
$(More\ f\ ss1\ C\ ss2)[t] = Fun\ f\ (ss1\ @\ C[t]\ \#\ ss2)$

For an empty context, the result is just the term t, whereas for a non-empty context, we recursively plug the term t into the subcontext C and put the result in between the left and the right argument lists. Finally, we apply the function symbol f to the resulting list of terms.

In examples, we again use the more compact notation and write, for example, $\mathsf{ack}(\Box, 0)$ instead of *More "ack"* [] $\Box$ [*Fun "0"* []].

Definition 4.9 (Composition of Contexts). In addition, contexts can be composed with each other by replacing the hole in the first context with the second context as follows:

$\Box \circ D = D$
$(More\ f\ ss\ C\ ts) \circ D = More\ f\ ss\ (C \circ D)\ ts$

Reasoning about Contexts.
As for terms, we do have case distinction and induction for contexts.

ctxt.exhaust: $[\![C = \Box \Longrightarrow P;\ \bigwedge f\ ss\ D\ ts.\ C = More\ f\ ss\ D\ ts \Longrightarrow P]\!] \Longrightarrow P$
ctxt.induct: $[\![P\ \Box;\ \bigwedge f\ ss\ D\ ts.\ P\ D \Longrightarrow P\ (More\ f\ ss\ D\ ts)]\!] \Longrightarrow P\ C$

This time, the automatically generated induction rule is exactly what we want. However, for case distinction, it is sometimes convenient to go 'inside out.'

ctxt_exhaust_rev:
$[\![C = \Box \Longrightarrow P;\ \bigwedge D\ f\ ss\ ts.\ C = D \circ (More\ f\ ss\ \Box\ ts) \Longrightarrow P]\!] \Longrightarrow P$

Further IsaFoR instantiates the type class *monoid_mult* with contexts, that is, it has been shown that contexts together with context composition and the unit element $\Box$, build a multiplicative monoid. Hence, all constants and lemmas that are available for *monoid_mult* inside Isabelle, can directly be used on contexts (for example, the n-fold composition of contexts).

4.4 Subterms

Having contexts, we can define the (*proper*) *subterm* relation $(\rhd)\ \trianglerighteq$.

Definition 4.10 (Subterm Relation). We say that t is a (proper) subterm of s, whenever there is a (non-empty) context C such that $s = C[t]$. That is,

$s \trianglerighteq t \equiv \exists\, C.\ s = C[t]$
$s \triangleright t \equiv \exists\, C.\ C \neq \Box \wedge s = C[t]$

To facilitate a reading from left to write, we sometimes refer to $\trianglerighteq$, as the *superterm* relation (since, we think that it is more natural to say that s is a superterm of t than that t is a subterm of s, when writing $s \trianglerighteq t$).

Reasoning about Subterms.
Using subterms, we can introduce an alternative induction scheme for terms, that is, induction over the proper subterms of a term:

subterm_induct: $(\bigwedge t.\ \forall\, s \lhd t.\ P\ s \Longrightarrow P\ t) \Longrightarrow P\ t$

4.5 Substitutions

Substitutions are needed to supply rules with concrete values to work on. In that sense, a substitution is just a function mapping variables to terms. In IsaFoR, we have

datatype (α, β) *subst* = *Subst* $(\beta \Rightarrow (\alpha, \beta)$ *term*$)$

For technical reasons we have to use a data type instead of a plain type abbreviation (otherwise it would not be possible to instantiate type classes by (α, β) *subst*).

In examples, we represent substitutions by finite sets of bindings. Then, $\{x/\mathsf{s}(y), z/u\}$ is a substitution that replaces every x by $\mathsf{s}(y)$, every z by u, and leaving all other variables unmodified.

Definition 4.11 (Substitutions on Terms). Applying a substitution to a term is defined by

$(Var\ x)\sigma = get\_subst\ \sigma\ x$
$(Fun\ f\ ts)\sigma = Fun\ f\ (map\ (\lambda t.\ t\sigma)\ ts)$

where *get_subst* extracts the actual variable mapping from a substitution, that is, *get_subst* $(Subst\ s) = s$.

Definition 4.12 (Composition of Substitutions). Also substitutions can be composed with each other. This is defined as follows:

$\sigma \circ \tau = Subst\ (\lambda x.\ (get\_subst\ \sigma\ x)\tau)$

Definition 4.13 (Substitutions on Contexts). Having substitutions on terms We can also apply a substitution to a context.

$\Box\sigma = \Box$
$(More\ f\ ss\ D\ ts)\sigma = More\ f\ (map\ (\lambda t.\ t\sigma)\ ss)\ D\sigma\ (map\ (\lambda t.\ t\sigma)\ ts)$

Reasoning about Substitutions.
As for contexts, IsaFoR contains an instantiation of substitutions to multiplicative monoids. This time, we have the neutral element

$subst\_empty\_def\colon \iota \equiv Subst\ Var$

(recall that the constructor *Var*, is nothing more than a function, taking some variable and building a term consisting solely of this variable out of it) and use the composition of substitutions as multiplication.

Another special substitution that is sometimes convenient, is one containing a singleton binding.

$subst\_def\colon \{x/t\} = Subst\ (\lambda y.\ \mathsf{if}\ x = y\ \mathsf{then}\ t\ \mathsf{else}\ Var\ y)$

Using substitutions, we can instantiate rules by specific values. For example, to compute the result of the function call $\mathsf{ack}(0, 0)$, we have to apply the substitution $\{y/0\}$ to the first rule of the *Ackermann* TRS.

In the next section, we define the formal semantics of TRSs.

4.6 Rewrite Relation

As mentioned above, the rules of a TRS can be seen as function definitions. To give semantics to a TRS, we would like to be able to compute the results of calling such functions. This is expressed by the *rewrite relation* $\to_{\mathcal{R}}$, induced by a TRS $\mathcal{R}$.

Definition 4.14 (Rewrite Relation). The rewrite relation is defined by closing the rules of a TRS under contexts and substitutions. Or equivalently, there is a *rewrite step* from s to t, denoted by $s \to_{\mathcal{R}} t$, whenever there exists a rule $(l, r) \in \mathcal{R}$, a context C, and a substitution σ, such that $s = C[l\sigma]$ and $t = C[r\sigma]$.

We write $s \to_{\mathcal{R}}^* t$ ($s \to_{\mathcal{R}}^+ t$), if there is a (non-empty) sequence of rewrite steps from s to t.

Hence, the semantics of a TRS $\mathcal{R}$ is given by the ARS represented by the relation $\to_{\mathcal{R}}$. Therefore, termination of a TRS $\mathcal{R}$ is defined as termination of the corresponding rewrite relation. For brevity, we write $\mathsf{SN}(\mathcal{R})$, instead of $\mathsf{SN}(\to_{\mathcal{R}})$.

Definition 4.15 (Non-Root / Root Rewrite Steps). For case distinctions it is often useful to differentiate between *root (rewrite) steps*, given by

$$\xrightarrow{\epsilon}_{\mathcal{R}} = \{(s,\, t) \mid \exists\, l\ r\ \sigma.\ (l,\, r) \in \mathcal{R} \wedge s = l\sigma \wedge t = r\sigma\}$$

and *non-root (rewrite) steps*, given by

$$\xrightarrow{>\epsilon}_{\mathcal{R}} = \{(s,\, t) \mid \exists\, l\ r\ C\ \sigma.\ (l,\, r) \in \mathcal{R} \wedge C \neq \Box \wedge s = C[l\sigma] \wedge t = C[r\sigma]\}$$

Note that $\to_{\mathcal{R}} \equiv \xrightarrow{\epsilon}_{\mathcal{R}} \cup \xrightarrow{>\epsilon}_{\mathcal{R}}$, since we just make sure that the involved context of a rewrite step is empty (for $\xrightarrow{\epsilon}_{\mathcal{R}}$) or non-empty (for $\xrightarrow{>\epsilon}_{\mathcal{R}}$). This is reflected in IsaFoR by the lemma

rstep_iff_rrstep_or_nrrstep: $\to_{\mathcal{R}} = \xrightarrow{\epsilon}_{\mathcal{R}} \cup \xrightarrow{>\epsilon}_{\mathcal{R}}$

Reasoning about the Rewrite Relation.
Internally, there are three (equivalent) characterizations of the rewrite relation. The original one is based on positions. Further, there is a characterization using two inductively defined sets: the closure under substitutions and the closure under contexts of a binary relation over terms. Finally, we have an inductively defined set *rstep* $\mathcal{R}$, given by the following rules:

rstep.id: $(s,\, t) \in \mathcal{R} \Longrightarrow s \to_{\mathcal{R}} t$
rstep.subst: $s \to_{\mathcal{R}} t \Longrightarrow s\sigma \to_{\mathcal{R}} t\sigma$
rstep.ctxt: $s \to_{\mathcal{R}} t \Longrightarrow C[s] \to_{\mathcal{R}} C[t]$

With hindsight, it turned out that the last characterization is most convenient for what we are presenting in the remainder. Hence, we do not give the details about the other two characterizations here. However, we believe, that positions are crucial as soon as we start to incorporate specific rewrite strategies.

Additionally to the above introduction rules, there is the following (combining all premises):

rstepI: $[\![(l,\, r) \in \mathcal{R};\ s = C[l\sigma];\ t = C[r\sigma]]\!] \Longrightarrow s \to_{\mathcal{R}} t$

Then, we have the elimination rule:

rstepE: $[\![s \to_{\mathcal{R}} t;\ \bigwedge C\ \sigma\ l\ r.\ [\![(l,\ r) \in \mathcal{R};\ s = C[l\sigma];\ t = C[r\sigma]]\!] \Longrightarrow P]\!] \Longrightarrow P$

Further, there are three case distinction rules. The first rule uses the restrictions of the rewrite relation to root steps and non-root steps.

rstep_cases: $[\![s \to_{\mathcal{R}} t;\ s \xrightarrow{\epsilon}_{\mathcal{R}} t \Longrightarrow P;\ s \xrightarrow{>\epsilon}_{\mathcal{R}} t \Longrightarrow P]\!] \Longrightarrow P$

The second rule is similar, but working directly on contexts:

rstep_cases′:
$[\![s \to_{\mathcal{R}} t;\ \bigwedge l\ r\ \sigma.\ [\![(l,\ r) \in \mathcal{R};\ l\sigma = s;\ r\sigma = t]\!] \Longrightarrow P;$

$\bigwedge f\ ss1\ u\ ss2\ v.$

$[\![s = Fun\ f\ (ss1\ @\ u\ \#\ ss2);\ t = Fun\ f\ (ss1\ @\ v\ \#\ ss2);\ u \to_{\mathcal{R}} v]\!] \Longrightarrow P]\!]$

$\Longrightarrow P$

The third rule is a refinement of the second one that does only work for well-formed TRSs:

rstep_cases_Fun′:
$[\![\mathsf{WF}(\mathcal{R});\ Fun\ f\ ss \to_{\mathcal{R}} t;$

$\bigwedge ls\ r\ \sigma.\ [\![(Fun\ f\ ls,\ r) \in \mathcal{R};\ map\ (\lambda t.\ t\sigma)\ ls = ss;\ r\sigma = t]\!] \Longrightarrow P;$

$\bigwedge i\ u.\ [\![i < |ss|;\ t = Fun\ f\ (take\ i\ ss\ @\ u\ \#\ drop\ (i+1)\ ss);\ ss_i \to_{\mathcal{R}} u]\!] \Longrightarrow P]\!]$

$\Longrightarrow P$

The reason is that for well-formed TRSs, rewrite steps can only start from non-variable terms (since all left-hand sides are guaranteed to be non-variable terms), thus having the shape *Fun f ss* for some function symbol f and some list of terms *ss*. There are two possibilities for the exact location of the rewrite step. Either it takes place at the root (and hence there is some rule $(Fun\ f\ ls,\ r)$ and a substitution σ, such that applying σ to all elements of *ls* results in *ss* and applying it to r, results in t. Or the rewrite step takes place below the root and hence only modifies one of the arguments in *ss*.

Finally, we have the induction rule:

rstep_induct: $[\![s \to_{\mathcal{R}} t;\ \bigwedge C\ \sigma\ l\ r.\ (l,\ r) \in \mathcal{R} \Longrightarrow P\ C[l\sigma]\ C[r\sigma]]\!] \Longrightarrow P\ s\ t$

Note that a TRS that is not well-formed is always nonterminating.

Lemma 4.1 (Well-Formedness is Necessary for Termination). *Let $\mathcal{R}$ be a TRS that is not well-formed. Then $\mathcal{R}$ is not terminating.*

This is easy to prove:

lemma *not_wf_trs_imp_not_SN_rstep*:
  assumes $\neg$ WF($\mathcal{R}$) **shows** $\neg$ SN($\mathcal{R}$)
proof −

If a TRS is not well-formed, then there is at least one rule where either the left-hand side is a variable or there is a variable in the right-hand side that does not occur in the left-hand side.

  from *assms* **obtain** l r **where** $(l, r) \in \mathcal{R}$
    and *bad_rule*: $l \in \mathcal{V} \vee (\exists x.\ x \in \mathcal{V}\mathsf{ar}(r) - \mathcal{V}\mathsf{ar}(l))$
      unfolding *wf_trs_def'* **by** *auto*
  from *bad_rule* **show** *?thesis*

We proceed by a case distinction, showing that in both cases, we can construct an infinite $\mathcal{R}$-sequence.

  proof

If the left-hand side is a variable x, an infinite sequence is obtained by iteratively stacking the substitution $\{x/r\}$ onto x.

    assume $l \in \mathcal{V}$
    then obtain x **where** l: $l = \mathit{Var}\ x$ **by** (*cases* l) *simp_all*
    let $?S = \lambda i.\ (\mathit{Var}\ x)(\{x/r\}^i)$
    have $\forall i.\ (?S\ i) \to_{\mathcal{R}} (?S\ (i+1))$ ⟨*proof omitted*⟩
    thus *?thesis* **by** *best*
  next

If there is a variable in the right-hand side that is not contained in the left-hand side, then we obtain an infinite $\mathcal{R}$-sequence by iteratively stacking the context $C\{x/l\}$ onto the left-hand side.

    assume $\exists x.\ x \in \mathcal{V}\mathsf{ar}(r) - \mathcal{V}\mathsf{ar}(l)$
    then obtain x **where** $x \in \mathcal{V}\mathsf{ar}(r) - \mathcal{V}\mathsf{ar}(l)$ **by** *auto*
    hence $r \trianglerighteq \mathit{Var}\ x$ **by** (*induct* r) *auto*

Here, we use the fact that whenever t is a subterm of s, then there is some context C, such that, $s = C[t]$:

supteqp_ctxt_E: $[\![s \trianglerighteq t; \bigwedge C.\ s = C[t] \Longrightarrow \mathit{thesis}]\!] \Longrightarrow \mathit{thesis}$

then obtain C **where** $r = C[\mathit{Var}\ x]$ **by** (*rule supteqp_ctxt_E*)
let $?S = \lambda i.\ ((C\{x/l\})^i)[l]$
have $\forall\, i.\ (?S\ i) \to_{\mathcal{R}} (?S\ (i+1))$ ⟨*proof omitted*⟩
thus *?thesis* **by** *best*
qed
qed

This shows that for termination analysis we may concentrate on well-formed TRSs. However, in the remainder we will explicitly state whenever we rely on well-formedness.

4.7 Chapter Notes

In this chapter, we have indicated why it is useful to have term structures for termination proofs. Then, we presented the basic definitions for term rewriting and several facts that are provided by IsaFoR to reason about terms and term rewriting.

A rather accessible introduction to term rewriting is given by Baader and Nipkow [2]. A more technical, but also more complete treatise is Terese [41].

Chapter 5

Dependency Pair Framework

Divide and conquer.

Design Principle

Every theory is a self-fulfilling prophecy that orders experience into the framework it provides.

Ruth Hubbard

Now that we have term rewriting as a simple yet powerful model of computation, we are interested in proving termination of TRSs automatically. To this end, an abundance of methods has been developed. First, *reduction orders*, that is, well-founded orders that embed the rules of a given TRS and thereby prove termination. Examples are: recursive path orders, the Knuth-Bendix order, and monotone algebras. Then, there are transformations on TRSs (like semantic labeling) that in many cases allow for 'simpler' termination proofs. And finally, there is the *dependency pair framework* that facilitates modular termination proofs using a combination of techniques. In this chapter, we concentrate on how to use the DP framework for proving termination of TRSs.

Clearly, a program is terminating if every sequence of consecutive function calls is finite. The same idea is underlying the DP framework, where for TRSs a 'function call' is just a subterm of some right-hand side, having a defined symbol as root (since only at such subterms, rewriting can potentially go on).

Example 5.1. Consider the following TRS, encoding addition on *Peano num-*

bers:

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

The only defined symbol is add. Observing the right-hand sides reveals that there is one subterm having add as root, namely, $\mathsf{add}(x, y)$. Now, consider the following rewrite sequence:

$$\mathsf{add}(\mathsf{add}(0, \mathsf{s}(0)), 0) \to \mathsf{add}(\mathsf{s}(0), 0) \to \mathsf{s}(\mathsf{add}(0, 0)) \to \mathsf{s}(0)$$

On a closer look, it can be seen that single steps are actually taking care of two things: executing a function (as in the last two steps) and evaluating arguments of a surrounding function (as in the first step; of course, evaluation of arguments also involves the execution of functions, but not of the surrounding one).

From a functional programming point of view, it is enough to show that every function inside a program is terminating (on arbitrary inputs) in order to conclude termination of the whole program. Furthermore, for proving termination of a single function on arbitrary inputs, it suffices to show its termination under the assumption that all of its arguments are terminating (since otherwise, the nonterminating argument would already give smaller evidence of nontermination of the whole program). This facilitates proofs by minimal counterexamples.

The same is true for term rewriting. Hence, in the next section, we concentrate on minimal counterexamples against termination.

5.1 Minimal Counterexamples

In term rewriting, a function call on some given input is represented by a term t. Additionally, the assumption that all arguments of the function are terminating, translates to the assumption that all proper subterms of t are terminating. For a given TRS $\mathcal{R}$ the set of all such *minimally nonterminating terms* is defined by

Tinf_def: $\quad \mathcal{T}^{\infty}_{\mathcal{R}} \equiv \{t \mid \neg\, \mathsf{SN}_{\mathcal{R}}(t) \wedge (\forall\, s \lhd t.\ \mathsf{SN}_{\mathcal{R}}(s))\}$

Note that for every nonterminating term t there is some subterm s of t, such that s is in $\mathcal{T}^\infty_\mathcal{R}$.

Lemma 5.1. *Let t be nonterminating with respect to $\mathcal{R}$. Then, there is a subterm s of t, such that s is minimally nonterminating with respect to $\mathcal{R}$.*

lemma *not_SN_imp_subt_Tinf*:
  assumes $\neg\ \mathsf{SN}_\mathcal{R}(t)$ **shows** $\exists\, s{\trianglelefteq}t.\ s \in \mathcal{T}^\infty_\mathcal{R}$

The proof uses induction over the subterms of t.

using *assms* **proof** (*induct rule*: *subterm_induct*)

From the induction hypothesis we know that t is not terminating and that every nonterminating proper subterm of t has itself a subterm that is in $\mathcal{T}^\infty_\mathcal{R}$. We proceed by a case distinction. Either all proper subterms of t are terminating (which means that t itself is minimally nonterminating), or there is a proper subterm of t that is nonterminating (in which case we can apply the induction hypothesis to finish the proof).

  case (*subterm t*) **show** *?case*
  proof (*cases* $\forall\, s{\triangleleft}t.\ \mathsf{SN}_\mathcal{R}(s)$)
    case *True* **with** *subterm* **show** *?thesis* **by** (*auto simp*: *Tinf_def*)
  next
    case *False*
    then obtain s **where** $t \rhd s$ **and** $\neg\ \mathsf{SN}_\mathcal{R}(s)$ **by** *best*
    with *subterm* **obtain** u **where** $u \trianglelefteq s$ **and** $u \in \mathcal{T}^\infty_\mathcal{R}$ **by** *auto*

There are several transitivity rules for $\rhd$ and $\trianglerighteq$. One of them is

  suptp_supteqp_trans: $[\![s \rhd t;\ t \trianglerighteq u]\!] \Longrightarrow s \rhd u$

    from $\langle t \rhd s\rangle$ **and** $\langle s \trianglerighteq u\rangle$ **have** $t \rhd u$ **by** (*rule suptp_supteqp_trans*)
    with $\langle u \in \mathcal{T}^\infty_\mathcal{R}\rangle$ **show** *?thesis* **by** *best*
  qed
qed

Having this, we can characterize termination of a TRS $\mathcal{R}$ by the emptiness of $\mathcal{T}^\infty_\mathcal{R}$, that is, $\mathcal{R}$ is nonterminating, if and only if, there is a term $t \in \mathcal{T}^\infty_\mathcal{R}$. (Note that '=' is also used for biimplication in HOL.)

Lemma 5.2. $(\mathcal{T}^\infty_\mathcal{R} = \varnothing) = \mathsf{SN}(\mathcal{R})$

Using Lemma 5.1, the proof is straight-forward.

Thus, we know that for every nonterminating TRS $\mathcal{R}$ there is a witness $t \in \mathcal{T}^\infty_\mathcal{R}$. By definition of $\mathcal{T}^\infty_\mathcal{R}$, all proper subterms of t are terminating. Further, we obtain an infinite rewrite sequence S, starting with t.

from ‹$t \in \mathcal{T}^\infty_\mathcal{R}$› **have** $\forall\, s \lhd t.\ \mathsf{SN}_\mathcal{R}(s)$ **and** $\neg\ \mathsf{SN}_\mathcal{R}(t)$ **by** (*auto simp*: *Tinf_def*)
then obtain S **where** $S\ 0 = t$ **and** *seq*: $\forall\, i.\ (S\ i) \to_\mathcal{R} (S\ (i+1))$ **by** *best*

Now, to distinguish between *executing a function* and *evaluating its arguments*, we use the two restrictions of the rewrite relation $\to_\mathcal{R}$ from Definition 4.15. Having this, together with $\forall\, s \lhd t.\ \mathsf{SN}_\mathcal{R}(s)$, implies that after a finite number of non-root steps, there has to follow a root step using some rule of $\mathcal{R}$, that is:

have $\exists\, i.\ (S\ 0) \xrightarrow{\geq\epsilon *}_\mathcal{R} (S\ i) \wedge (S\ i) \xrightarrow{\epsilon}_\mathcal{R} (S\ (i+1))$
proof −
  from ‹$t \in \mathcal{T}^\infty_\mathcal{R}$› **have** $\mathsf{SN}_{\to^{\geq\epsilon}_\mathcal{R}}(S\ 0)$
    by (*simp add*: *Tinf_imp_SN_elt_nrrstep* ‹$S\ 0 = t$›)
  with *seq* **and** *union_iseq_SN_elt_imp_first_step*[*of* $S\ \xrightarrow{\epsilon}_\mathcal{R}\ \xrightarrow{\geq\epsilon}_\mathcal{R}$]
  obtain j **where** *first_step*: $(S\ j) \xrightarrow{\epsilon}_\mathcal{R} (S\ (j+1))$
    and *nrsteps*: $\forall\, i{<}j.\ (S\ i) \xrightarrow{\geq\epsilon}_\mathcal{R} (S\ (i+1))$
    by (*auto simp*: *rstep_iff_rrstep_or_nrrstep*)
  from *nrsteps* **have** $(S\ 0) \xrightarrow{\geq\epsilon *}_\mathcal{R} (S\ j)$ **by** (*induct* j) *auto*
  with *first_step* **show** *?thesis* **by** *auto*
qed
then obtain j **where** *nrseq*: $(S\ 0) \xrightarrow{\geq\epsilon *}_\mathcal{R} (S\ j)$
  and *rrstep*: $(S\ j) \xrightarrow{\epsilon}_\mathcal{R} (S\ (j+1))$ **by** *auto*
obtain $l\ r\ \sigma$ **where** $(l,\ r) \in \mathcal{R}$ **and** $l\sigma = S\ j$ **and** $r\sigma = S\ (j+1)$ **by** *auto*

We refer to Appendix A for the proofs of the lemmas *Tinf_imp_SN_elt_nrrstep* and *union_iseq_SN_elt_imp_first_step*.

Clearly, $r\sigma$ is not terminating, since it is part of the infinite sequence S. Additionally, every term that is nonterminating has some subterm which is minimally nonterminating, due to Lemma 5.1. Thus, there is a corresponding subterm t' of $r\sigma$.

have $\neg\ \mathsf{SN}_\mathcal{R}(r\sigma)$ ⟨*proof omitted*⟩

then obtain t' **where** $r\sigma \trianglerighteq t'$ **and** $t' \in \mathcal{T}^\infty_\mathcal{R}$
  using *not_SN_imp_subt_Tinf* **by** *auto*

At this point we analyze t further. Remember that for TRSs which are not well-formed, nontermination is immediate. Hence, we restrict our investigations to well-formed TRSs. But then, only non-variable terms can be members of $\mathcal{T}^\infty_\mathcal{R}$, since variables are normal forms.

have $\exists f\ ts.\ t = Fun\ f\ ts$ ⟨*proof omitted*⟩
then obtain $f\ ts$ **where** t: $t = Fun\ f\ ts$ (**is** $t = ?t$) **by** *auto*

Hence, the rewrite sequence from $S\ 0$ to $S\ j$ starts with the non-variable term $Fun\ f\ ts$. Since non-root steps obviously preserve the root, this implies that $S\ j$ is a function application of f to some argument list ls'.

from *nrseq* **have** $?t \xrightarrow{>\epsilon}{}^*_\mathcal{R}\ (S\ j)$ **by** (*simp add*: t ‹$S\ 0 = t$›)
moreover from *nrrsteps_preserve_root*[*OF this*] **obtain** ls'
  where $S\ j = Fun\ f\ ls'$ **by** *auto*

Combining the above with the fact that $l\sigma = S\ j$, results in a non-root rewrite sequence from $Fun\ f\ ts$ to $l\sigma$.

ultimately
  have $l\sigma = Fun\ f\ ls'$ **and** $?t \xrightarrow{>\epsilon}{}^*_\mathcal{R}\ (Fun\ f\ ls')$ **by** (*auto simp*: ‹$l\sigma = S\ j$›)
hence $?t \xrightarrow{>\epsilon}{}^*_\mathcal{R}\ l\sigma$ **by** *simp*

Since $\mathcal{R}$ is well-formed, we know that l is a function application of some function symbol g to some argument list ls.

from *wf_trs_imp_lhs_Fun*[*OF* ‹*wf_trs* $\mathcal{R}$› ‹$(l, r) \in \mathcal{R}$›] **obtain** $g\ ls$
  where $l = Fun\ g\ ls$ **by** *best*

Clearly, instantiating any argument of l with σ, results in the corresponding argument of $l\sigma$.

with ‹$l\sigma = Fun\ f\ ls'$› **have** $\forall\, i{<}|ls'|.\ \ ls'_i = (ls_i)\sigma$ **by** *auto*

By the fact that all previous steps are below the root, we know that every argument of $?t$, rewrites to the corresponding argument of the instantiated left-hand side l.

have $\forall\, i{<}|ts|.\ \ ts_i \to_{\mathcal{R}}^* (ls_i)\sigma$ ⟨*proof omitted*⟩

Obviously, those steps do not change the number of arguments (since all of them take place below the root) and thus, we may use $|ts|$ and $|ls'|$ interchangeably.

from ⟨$?t \overset{\trianglerighteq\epsilon}{\to}_{\mathcal{R}}^* l\sigma$⟩ **have** $|ts| = num\_args\ l\sigma$
  by (*induct*) (*auto simp*: *nrrstep_equiv_num_args*)
with ⟨$l\sigma = Fun\ f\ ls'$⟩ **and** ⟨$\forall\, i{<}|ls'|.\ \ ls'_i = ls_i\sigma$⟩
  have $\forall\, i{<}|ts|.\ \ ls_i\sigma = ls'_i$ **by** *simp*

Now, since all proper subterms of t are terminating and rewriting preserves termination, we know that all arguments of $l\sigma$ are terminating.

moreover with ⟨$\forall\, s{\lhd}t.\ \mathsf{SN}_{\mathcal{R}}(s)$⟩ **have** $\forall\, i{<}|ts|.\ \ \mathsf{SN}_{\mathcal{R}}(ts_i)$ **by** (*auto simp*: *t*)
ultimately have $\forall\, i{<}|ts|.\ \ \mathsf{SN}_{\mathcal{R}}(ls_i\sigma)$
  using ⟨$\forall\, i{<}|ts|.\ \ ts_i \to_{\mathcal{R}}^* ls_i\sigma$⟩
  and *steps_preserve_SN_elt*[*of* _ _ $\to_{\mathcal{R}}$] **by** *blast*
with ⟨$\forall\, i{<}|ts|.\ \ ls_i\sigma = ls'_i$⟩ **and** ⟨$|ts| = num\_args\ l\sigma$⟩
  and ⟨$l\sigma = Fun\ f\ ls'$⟩ **and** ⟨$\forall\, i{<}|ts|.\ \mathsf{SN}_{\mathcal{R}}(ls_i\sigma)$⟩
  have $\forall\, i{<}num\_args\ (Fun\ f\ ls').\ \mathsf{SN}_{\mathcal{R}}(ls'_i)$ **by** *auto*

This implies that all proper subterms of $l\sigma$ are terminating.

hence $\forall\, s{\lhd}Fun\ f\ ls'.\ \mathsf{SN}_{\mathcal{R}}(s)$ **using** *SN_args_imp_SN_subt*[*of f ls'* $\mathcal{R}$] **by** *blast*
hence $\forall\, s{\lhd}l\sigma.\ \mathsf{SN}_{\mathcal{R}}(s)$ **by** (*simp add*: ⟨$l\sigma = Fun\ f\ ls'$⟩)

Since $l\sigma$ is an element of the sequence S, it is clearly not terminating. Together with the fact that $\forall\, s{\lhd}l\sigma.\ \mathsf{SN}_{\mathcal{R}}(s)$, this implies that $l\sigma$ is minimally nonterminating.

have $l\sigma \in \mathcal{T}_{\mathcal{R}}^{\infty}$ ⟨*proof omitted*⟩

moreover from ⟨$(l,\ r) \in \mathcal{R}$⟩ **have** $l\sigma \overset{\epsilon}{\to}_{\mathcal{R}} r\sigma$ **unfolding** *rrstep_def'* **by** *best*

Next, we show that there is a non-variable subterm u of r, such that its instantiation with σ is itself minimally nonterminating. Additionally, we show that $u\sigma$ cannot be a proper subterm of the instantiated left-hand side $l\sigma$ (this is done, in order to obtain a stronger definition of dependency pairs; cf. Definition 5.1).

moreover have $\exists\, u.\ r \trianglerighteq u \wedge u \notin \mathcal{V} \wedge r\sigma \trianglerighteq u\sigma \wedge u\sigma \in \mathcal{T}_{\mathcal{R}}^{\infty} \wedge l\sigma \not\trianglerighteq u\sigma$

proof −

We start by noting that σ-instances of variables in the right-hand side r, are terminating (otherwise, we would have a contradiction to the fact $\forall\, i{<}|ts|.\ \mathsf{SN}_{\mathcal{R}}(ls_i\sigma)$).

have $\forall\, x{\in}\mathcal{V}\mathsf{ar}(r).\ \mathsf{SN}_{\mathcal{R}}((\mathit{Var}\ x)\sigma)$
proof
fix x **assume** $x \in \mathcal{V}\mathsf{ar}(r)$
hence $x \in \mathcal{V}\mathsf{ar}(l)$ **using** ‹$(l,\ r) \in \mathcal{R}$› **and** ‹*wf_trs* $\mathcal{R}$›
by (*auto simp*: *wf_trs_def*)
hence $l \rhd \mathit{Var}\ x$ **by** (*auto simp*: ‹$l = \mathit{Fun}\ g\ ls$› *intro*: *var_imp_supteqp*)
hence $l\sigma \rhd (\mathit{Var}\ x)\sigma$ **by** (*rule suptp_stable*)
with ‹$l\sigma \in \mathcal{T}^{\infty}_{\mathcal{R}}$›
show $\mathsf{SN}_{\mathcal{R}}((\mathit{Var}\ x)\sigma)$ **unfolding** *Tinf_def* **by** *simp*
qed

But then, since $t' \in \mathcal{T}^{\infty}_{\mathcal{R}}$, it is impossible for t' to be a subterm of a σ-instance of some variable in r.

have $\forall\, x{\in}\mathcal{V}\mathsf{ar}(r).\ (\mathit{Var}\ x)\sigma \not\trianglerighteq t'$
proof
fix x **assume** $x \in \mathcal{V}\mathsf{ar}(r)$
hence $\mathsf{SN}_{\mathcal{R}}((\mathit{Var}\ x)\sigma)$ **using** ‹$\forall\, x{\in}\mathcal{V}\mathsf{ar}(r).\ \mathsf{SN}_{\mathcal{R}}((\mathit{Var}\ x)\sigma)$› **by** *simp*
show $(\mathit{Var}\ x)\sigma \not\trianglerighteq t'$
proof
assume $(\mathit{Var}\ x)\sigma \trianglerighteq t'$
with ‹$\mathsf{SN}_{\mathcal{R}}((\mathit{Var}\ x)\sigma)$› **have** $\mathsf{SN}_{\mathcal{R}}(t')$ **using** *SN_imp_SN_subt* **by** *auto*
with ‹$t' \in \mathcal{T}^{\infty}_{\mathcal{R}}$› **show** *False* **unfolding** *Tinf_def* **by** *auto*
qed
qed

We know, however, that t' is a subterm of $r\sigma$. Thus, there is some subterm u of r, such that, $t' = u\sigma$.

from *subt_instance_and_not_subst_imp_subt*[*OF* ‹$r\sigma \trianglerighteq t'$› *this*]
obtain u **where** $r \trianglerighteq u$ **and** $t' = u\sigma$ **by** *best*

Moreover, u cannot be a variable, since this would directly contradict to

$\forall\, x{\in}\mathcal{V}\mathsf{ar}(r).\ \mathsf{SN}_{\mathcal{R}}((\mathit{Var}\ x)\sigma)$.
moreover have $u \notin \mathcal{V}$
proof
assume $u \in \mathcal{V}$
then obtain x **where** $u = \mathit{Var}\ x$ **by** (*cases* u) *auto*
with ‹$r \trianglerighteq u$› **have** $r \trianglerighteq \mathit{Var}\ x$ **by** *simp*
from *var_subt_imp_subt_in_vars*[*OF this*] **have** $x \in \mathcal{V}\mathsf{ar}(r)$ **.**
with ‹$\forall\, x{\in}\mathcal{V}\mathsf{ar}(r).\ \mathsf{SN}_{\mathcal{R}}((\mathit{Var}\ x)\sigma)$› **and** ‹$t' \in \mathcal{T}^{\infty}_{\mathcal{R}}$›
show *False* **by** (*simp add*: *Tinf_def* ‹$u = \mathit{Var}\ x$› ‹$t' = u\sigma$›)
qed

moreover from ‹$r \trianglerighteq u$› **have** $r\sigma \trianglerighteq u\sigma$ **by** (*rule supteqp_stable*)
moreover from ‹$t' \in \mathcal{T}^{\infty}_{\mathcal{R}}$› **have** $u\sigma \in \mathcal{T}^{\infty}_{\mathcal{R}}$ **by** (*simp add*: ‹$t' = u\sigma$›)
moreover have $l\sigma \not\trianglerighteq u\sigma$
proof
assume $l\sigma \triangleright u\sigma$
with ‹$l\sigma \in \mathcal{T}^{\infty}_{\mathcal{R}}$› **have** $\mathsf{SN}_{\mathcal{R}}(u\sigma)$ **unfolding** *Tinf_def* **by** *force*
with ‹$u\sigma \in \mathcal{T}^{\infty}_{\mathcal{R}}$› **show** *False* **by** (*simp add*: *Tinf_def*)
qed
ultimately show *?thesis* **by** *auto*
qed
moreover from ‹$(l,\ r) \in \mathcal{R}$›
have $l\sigma \xrightarrow{\epsilon}_{\mathcal{R}} r\sigma$ **unfolding** *rrstep_def'* **by** *best*
ultimately
show *?thesis* **using** ‹$(l,\ r) \in \mathcal{R}$› **and** ‹$?t \xrightarrow{\trianglerighteq\epsilon *}_{\mathcal{R}} l\sigma$› **by** (*auto simp*: t)
qed

Putting all this together we obtain the following lemma:

Lemma 5.3. *Let $\mathcal{R}$ be a well-formed TRS. Then, for every term $t \in \mathcal{T}^{\infty}_{\mathcal{R}}$, there exist a rewrite rule $(l,\ r) \in \mathcal{R}$, a substitution σ, and a non-variable subterm u of r, such that $t \xrightarrow{\trianglerighteq\epsilon *}_{\mathcal{R}} l\sigma$, $l\sigma \xrightarrow{\epsilon}_{\mathcal{R}} r\sigma$, and $r\sigma \trianglerighteq u\sigma$. Further, $l\sigma \in \mathcal{T}^{\infty}_{\mathcal{R}}$, $u\sigma \in \mathcal{T}^{\infty}_{\mathcal{R}}$, and $u\sigma$ is not a proper subterm of $l\sigma$.*

In IsaFoR this is stated as:

$[\![\mathsf{WF}(\mathcal{R});\ t \in \mathcal{T}^{\infty}_{\mathcal{R}}]\!] \Longrightarrow \exists\, l\ r\ u\ \sigma.\ (l,\ r) \in \mathcal{R} \wedge r \trianglerighteq u \wedge u \notin \mathcal{V} \wedge t \xrightarrow{\trianglerighteq\epsilon *}_{\mathcal{R}} l\sigma \wedge l\sigma \xrightarrow{\epsilon}_{\mathcal{R}} r\sigma \wedge r\sigma \trianglerighteq u\sigma \wedge l\sigma \in \mathcal{T}^{\infty}_{\mathcal{R}} \wedge u\sigma \in \mathcal{T}^{\infty}_{\mathcal{R}} \wedge l\sigma \not\trianglerighteq u\sigma$

This corresponds to Lemma 1 of Hirokawa and Middeldorp [17]. By using Lemma 5.1, followed by repeated applications of Lemma 5.3, every nonterminating term t, gives rise to an infinite sequence of the shape:

$$t = t_1 \trianglerighteq u_1 \overset{>\epsilon *}{\to}_{\mathcal{R}} s_1 \overset{\epsilon}{\to}_{\mathcal{R}} t_2 \trianglerighteq u_2 \overset{>\epsilon *}{\to}_{\mathcal{R}} s_2 \overset{\epsilon}{\to}_{\mathcal{R}} t_3 \trianglerighteq u_3 \overset{>\epsilon *}{\to}_{\mathcal{R}} s_3 \overset{\epsilon}{\to}_{\mathcal{R}} \cdots$$

Our next goal is to simplify such a sequence by introducing a new TRS, the so called *dependency pairs* of $\mathcal{R}$. This allows to get rid of any positioning constraints and subterm steps, resulting in

$$f(u_1) \to_{\mathcal{R}}^* f(s_1) \to_{\mathsf{DP}(\mathcal{R})} f(u_2) \to_{\mathcal{R}}^* f(s_2) \to_{\mathsf{DP}(\mathcal{R})} f(u_3) \to_{\mathcal{R}}^* f(s_3) \to_{\mathsf{DP}(\mathcal{R})} \cdots$$

where f is some way of marking the root symbol of a term, such that $\mathcal{R}$-steps at the root are no longer possible. The dependency pairs of a TRS are defined in the next section.

5.2 Dependency Pairs

In IsaFoR, we need a concise notion of 'marking.' Moreover, since the type of terms is parametrized by the type of function symbols, we cannot assume any properties on them (like, *a function symbol is a lowercase string*). Thus, we introduce a new type, denoting (potentially) marked symbols:

datatype α *shp* = *Sharp* α | *Plain* α

The name *shp*, stands for 'sharp,' and was chosen since in the literature, marking is usually done using a sharp symbol ($\sharp$), and sometimes called *sharping*. The type has two constructors, one (*Sharp*) for sharped function symbols and another (*Plain*) for function symbols that are not marked (this is necessary since all function symbols in a term, have to be of the same type).

Marking a term is done by sharping its root. To do that, we need an auxiliary function *plain*::(α, β) *term* $\Rightarrow$ (α *shp*, β) *term*, lifting unsharped terms into the type of sharped terms.

plain (*Var* x) = *Var* x
plain (*Fun* f *ts*) = *Fun* (*Plain* f) (*map plain ts*)

$(Var\ x)^\sharp$ = *Var* x
$(Fun\ f\ ts)^\sharp$ = *Fun* (*Sharp* f) (*map plain ts*)

Now we can define the dependency pairs of a TRS $\mathcal{R}$ as follows:

Definition 5.1 (Dependency Pairs). Given a TRS $\mathcal{R}$, the set of its *dependency pairs* $\mathsf{DP}(\mathcal{R})$ is given by

$$\mathsf{DP}(\mathcal{R}) = \{(s,\ t) \mid \exists\, l\ r\ f\ u.\ (l,\ r) \in \mathcal{R} \wedge s = l^\sharp \wedge t = u^\sharp \wedge r \trianglerighteq u \wedge \mathit{root}\ u = \mathit{Some}\ f \wedge (f,\ \mathit{num\_args}\ u) \in \mathcal{D}_\mathcal{R} \wedge l \not\trianglerighteq u\}$$

Note that this definition includes an improvement due to Dershowitz [9]: We may safely ignore subterms of right-hand sides that are proper subterms of the corresponding left-hand side. (This was already accounted for in Lemma 5.3.) Further note that in examples we capitalize function symbols in order to indicate sharping and thus write for example ADD instead of *Sharp* add.

Example 5.2. For the TRS of Example 5.1 we obtain the single dependency pair

$$\mathsf{ADD}(\mathsf{s}(x), y) \to \mathsf{ADD}(x, y)$$

reflecting its recursive call-structure.

When the original TRS has the type $(\alpha,\ \beta)\ \mathit{trs}$, then its dependency pairs have the type $(\alpha\ \mathit{shp},\ \beta)\ \mathit{trs}$. That means that typewise, rules from $\mathsf{DP}(\mathcal{R})$ are incompatible with rules from $\mathcal{R}$. For that reason, we need a lifting of the TRS $\mathcal{R}$ into the proper type (using the *plain*-function from above):

$$\mathsf{ID}(\mathcal{R}) = \{(s,\ t) \mid \exists\, l\ r.\ s = \mathit{plain}\ l \wedge t = \mathit{plain}\ r \wedge (l,\ r) \in \mathcal{R}\}$$

At this point, we have the pair of TRSs $(\mathsf{DP}(\mathcal{R}), \mathsf{ID}(\mathcal{R}))$ at our disposal, where the first component contains all the function calls of which we want to prove termination, and the second component provides the rules for evaluating function arguments. Such a pair of TRSs, is called a *dependency pair problem*. The obvious question is: *How is a dependency pair problem related to the termination of a term rewrite system?* We give an answer in the next section.

5.2.1 Termination of TRSs using Dependency Pairs

In this section, we show that a minimal counterexample for termination is equivalent to a *minimal infinite chain* of $\mathsf{DP}(\mathcal{R})$-steps relative to $\to^*_{\mathsf{ID}(\mathcal{R})}$. The intuition is that every $\mathsf{DP}(\mathcal{R})$-step corresponds to a (recursive) function call, whereas an $\mathsf{ID}(\mathcal{R})$ sequence denotes evaluating the arguments of a function.

Before we formally state the main theorem of this section, we give some necessary definitions.

Definition 5.2 (DP Problems). A *dependency pair problem* is a pair of TRSs $(\mathcal{P}, \mathcal{R})$. In IsaFoR, we use the type abbreviation

types (α, β) dpp $= (\alpha, \beta)$ trs $\times$ (α, β) trs

Definition 5.3 (Infinite Chains). An *infinite* $(\mathcal{P}, \mathcal{R})$*-chain* is defined by

$$\mathsf{ichain}\ (\mathcal{P}, \mathcal{R})\ s\ t\ \sigma \equiv \forall i.\ (s\ i,\ t\ i) \in \mathcal{P} \wedge (t\ i)(\sigma\ i) \rightarrow^*_{\mathcal{R}} (s\ (i+1))(\sigma\ (i+1))$$

where s and t are infinite sequences of terms, and σ is an infinite sequence of substitutions.

Note that this definition of infinite chains is slightly different from the original one, given by Arts and Giesl [1]. Instead of demanding a single substitution that is used for all $\mathsf{ID}(\mathcal{R})$-sequences between terms $t\ i$ and $s\ (i+1)$, as in [1], we use an infinite sequence of substitutions. This avoids reasoning about fresh variables and substitutions with infinite domains in the formalization of infinite chains. (Note that IsaFoR's definition of substitution allows infinite domains—depending on the type of variables that is used. However, the advantage here, is that we do not have to reason under the assumption of having an infinite domain.)

Note that the right-hand sides of $\mathsf{DP}(\mathcal{R})$ have exactly the shape of the term u from Lemma 5.3. In fact, since we are just interested in well-formed TRSs, every term in $\mathcal{T}^\infty_{\mathcal{R}}$ has a defined root (with respect to $\mathcal{R}$), that is,

$$[\![\mathsf{WF}(\mathcal{R});\ t \in \mathcal{T}^\infty_{\mathcal{R}}]\!] \Longrightarrow (\mathsf{root}(t),\ \mathit{num\_args}\ t) \in \mathcal{D}_{\mathcal{R}}$$

This implies that the right-hand sides of the dependency pairs cover all possible us (as given by Lemma 5.3). It still remains to lift this observation to the proper types.

Lemma 5.4. $[\![\mathsf{WF}(\mathcal{R});\ t \in \mathcal{T}^\infty_{\mathcal{R}}]\!] \Longrightarrow \exists\, l\ r\ u\ \sigma.\ t^\sharp \rightarrow^*_{\mathsf{ID}(\mathcal{R})} l\sigma \wedge (l,\ r) \in \mathsf{DP}(\mathcal{R}) \wedge r\sigma = u^\sharp \wedge u \in \mathcal{T}^\infty_{\mathcal{R}}$

We see that for every minimally nonterminating term, after a finite number of rewrite steps starting from its sharped version, there is a step (using a dependency pair), such that the resulting term is again minimally nonterminating.

This, together with the definition of infinite chains, results in the notion of minimal infinite chains.

Definition 5.4 (Minimal Infinite Chains). A *minimal infinite* $(\mathcal{P}, \mathcal{R})$*-chain* is defined by

$$\mathsf{ichain}_{\mathsf{min}}\ (\mathcal{P},\ \mathcal{R})\ s\ t\ \sigma \equiv \mathsf{ichain}\ (\mathcal{P},\ \mathcal{R})\ s\ t\ \sigma \wedge (\forall\, i.\ \mathsf{SN}_{\mathcal{R}}((t\ i)(\sigma\ i)))$$

for the infinite sequences of terms s and t, and the infinite sequence of substitutions σ.

The only difference between minimal infinite chains and infinite chains, is that the former demand all the intermediate $(t\ i)(\sigma\ i)$ to be terminating with respect to $\mathcal{R}$. Finally, we are in a position where we can state the crucial connection between the termination of a TRS $\mathcal{R}$ and the corresponding DP problem $(\mathsf{DP}(\mathcal{R}),\ \mathsf{ID}(\mathcal{R}))$. Whenever $\mathcal{R}$ is not terminating, then there is a minimal infinite $(\mathsf{DP}(\mathcal{R}),\ \mathsf{ID}(\mathcal{R}))$-chain.

Theorem 5.1 (Nontermination implies Minimal Chains).
$[\![\mathsf{WF}(\mathcal{R});\ \neg\ \mathsf{SN}(\mathcal{R})]\!] \Longrightarrow \exists\, s\ t\ \sigma.\ \mathsf{ichain}_{\mathsf{min}}\ (\mathsf{DP}(\mathcal{R}),\ \mathsf{ID}(\mathcal{R}))\ s\ t\ \sigma$

Here, $(\mathsf{DP}(\mathcal{R}),\ \mathsf{ID}(\mathcal{R}))$ is sometimes called the *initial DP problem.*

Example 5.3. Recall Examples 5.1 and 5.2, resulting in the initial DP problem:

$$\left(\{\mathsf{ADD}(\mathsf{s}(x), y) \to \mathsf{ADD}(x, y)\},\ \left\{\begin{array}{c}\mathsf{add}(\mathsf{0}, y) \to y,\\ \mathsf{add}(\mathsf{s}(x), y) \to \mathsf{s}(\mathsf{add}(x, y))\end{array}\right\}\right)$$

5.3 Finiteness and Processors

In this section we introduce the dependency pair framework, as formalized in IsaFoR. In the DP framework, we are concerned with the 'termination behavior' of DP problems, where the 'termination behavior' of a DP problem is characterized by the existence of minimal infinite chains.

Definition 5.5 (Finiteness of DP Problems). We say that a dependency pair problem $(\mathcal{P},\ \mathcal{R})$ is *finite* whenever it does not allow any minimal infinite chains, that is,

$$\mathsf{finite}(\mathcal{P},\,\mathcal{R}) \equiv \nexists\, s\ t\ \sigma.\ \mathsf{ichain}_{\mathsf{min}}\ (\mathcal{P},\,\mathcal{R})\ s\ t\ \sigma$$

The main advantage of using the DP framework over direct termination proofs for TRSs is that there are stronger termination techniques for showing finiteness of DP problems. To name a few of those so called *(DP) processors*: there is the subterm criterion and usable rules, both relying on the minimality of infinite chains; then there is the decomposition of the dependency graph, splitting a single DP problem (possibly) into several smaller ones; further, there are weakly monotone reduction pairs, which have weaker constraints than their strictly monotone counterparts in the direct setting.

Definition 5.6 (Sound DP Processors). From the perspective of a termination tool, a *dependency pair processor* (or *processor* for short) is a function taking a DP problem $(\mathcal{P},\,\mathcal{R})$ as input and returning either "no" (indicating that $(\mathcal{P},\,\mathcal{R})$ is *not* finite) or a set of new DP problems. A processor is *sound*, whenever finiteness of all the generated DP problems implies finiteness of the given DP problem. Hence, sound processors may be composed for proving finiteness of a DP problem. In contrast to a termination tool, in our development, we do not have to compute the result of applying a processor to a DP problem, but rather have to check whether some given result really constitutes a correct application of a processor. Thus, we model a processor by a function p::$[(\alpha,\,\beta)\ \mathit{trs},\,(\alpha,\,\beta)\ \mathit{trs},\,(\alpha,\,\beta)\ \mathit{trs},\,(\alpha,\,\beta)\ \mathit{trs}] \Rightarrow \mathit{bool}$, such that $p\ \mathcal{P}\ \mathcal{R}\ \mathcal{Q}\ \mathcal{S}$ is *True*, whenever the pair $(\mathcal{Q},\,\mathcal{S})$ is in the result of applying the processor to the DP problem $(\mathcal{P},\,\mathcal{R})$. (Note that this is just a different, but equivalent, formulation. It does not restrict our results in any way.) Then soundness of p is defined by

$$\mathsf{sound}\ p \equiv \forall\,\mathcal{P}\ \mathcal{R}\ \mathcal{Q}\ \mathcal{S}.\ p\ \mathcal{P}\ \mathcal{R}\ \mathcal{Q}\ \mathcal{S} \wedge \mathsf{finite}(\mathcal{Q},\,\mathcal{S}) \longrightarrow \mathsf{finite}(\mathcal{P},\,\mathcal{R})$$

By Theorem 5.1, a TRS that is nonterminating has a minimal infinite chain with respect to its initial DP problem. Put differently, if the initial DP problem of a TRS is finite, then the TRS is terminating. Thus, for proving termination of a TRS, we try to prove finiteness of its initial DP problem. This is done by recursively applying sound processors, thereby generating a tree of DP problems. A branch of this tree stops to grow, as soon as an applied processor results in an empty set of new DP problems. At the end, we have a rooted tree where all the leaves are finite DP problems, and by the soundness of all the processors that are applied at non-leaf nodes, finiteness propagates up to the root—the initial DP problem.

5.4 Chapter Notes

The DP framework was introduced by Giesl et al. [12] as a generalization of the DP approach [1]. In its most general form, it is concerned with $\mathcal{Q}$-restricted rewriting, unifying techniques for full termination (that is, without a specific evaluation strategy) and innermost rewriting. A simpler variant that just deals with full rewriting (and is very similar to our Isabelle formalization of the DP framework) can be found in [14].

Chapter 6

Subterm Criterion

Keep it short and simple.

Design Principle

In the last chapter, we gave our formalization of the dependency pair framework. Thus, setting the stage for the automatic certification of termination proofs. However, we are still lacking (sound) processors that can be applied inside this framework. In the following we present the *subterm criterion*, a processor that is especially nice, due to its simplicity.

Example 6.1. Taking the DP problem from Example 5.3, it can be seen that the first argument of ADD does strictly decrease (in the sense that the first argument of the right-hand side is a proper subterm of the first argument of the left-hand side) in every recursive call. We will show that this is enough to conclude finiteness of the DP problem (and thus termination of the original system).

The subterm criterion covers exactly this kind of reasoning. We shall start by presenting the subterm criterion in its original version. Then we reformulate it as a processor in the DP framework. Afterwards we describe a generalized version of the subterm criterion, due to a Coq formalization which was obtained independently from our efforts in Isabelle. Finally, we give our formalization in IsaFoR and conclude with some remarks on the practical applicability of the generalized version.

6.1 Original Version

When Hirokawa and Middeldorp [17, 18] first introduced the subterm criterion, the DP framework did not exist. Hence, the criterion covers more than it actually needs, if formulated as a DP processor. More specifically, in its original version, the subterm criterion is concerned with cycles in the so called *dependency graph* of a TRS. (Actually, also an optimization was introduced, using strongly connected components instead of cycles and decomposing problems recursively. This can be seen as the basis of the DP framework.) To appreciate this formulation, we shall shortly present what this dependency graph actually is.

Definition 6.1 (Dependency Graph). Recall the dependency pairs of a TRS $\mathcal{R}$ (given by Definition 5.1). As already mentioned earlier, those encode the possible (recursive) calls of $\mathcal{R}$. However, the DPs do not restrict the order in which such calls may actually occur inside a rewrite sequence (that is, the mutual dependencies of the recursive functions encoded by $\mathcal{R}$). This is handled by the *dependency graph* $\mathsf{DG}(\mathcal{R})$ of $\mathcal{R}$, which is a directed graph whose vertices are $\mathsf{DP}(\mathcal{R})$ and there is an edge from $(s,t) \in \mathsf{DP}(\mathcal{R})$ to $(u,v) \in \mathsf{DP}(\mathcal{R})$, if and only if there exist substitutions σ and τ such that

$$t\sigma \to_{\mathcal{R}}^{*} u\tau$$

(Note that the dependency graph is in general not computable. Hence, in practice approximations are used. This will, however, not influence our discussion.)

The original version of the subterm criterion is as follows:

Theorem 6.1 (Hirokawa and Middeldorp [18], Theorem 8). *Let $\mathcal{R}$ be a TRS and let $\mathcal{C}$ be a cycle in $\mathsf{DG}(\mathcal{R})$. If there exists a simple projection π for $\mathcal{C}$ such that $\pi(\mathcal{C}) \subseteq \trianglerighteq$ and $\pi(\mathcal{C}) \cap \triangleright \neq \varnothing$ then there are no $\mathcal{C}$-minimal rewrite sequences.*

In the above statement, a simple projection π is a function that maps a function application to one of its arguments, depending on the term's root symbol. Further, a $\mathcal{C}$-minimal rewrite sequence corresponds to a minimal infinite $(\mathcal{C},\mathcal{R})$-chain. (Since for a finite TRS $\mathcal{R}$, we have a finite dependency graph and thus, every infinite rewrite sequence eventually stays inside some cycle of

$\mathsf{DG}(\mathcal{R})$, we may prove termination by showing that none of $\mathsf{DG}(\mathcal{R})$'s cycles $\mathcal{C}$, allows a $\mathcal{C}$-minimal rewrite sequence.)

6.2 Subterm Criterion Processor

By reformulating the same criterion as a DP processor (as for example already done by Thiemann [42]), we roughly arrive at the following statement:

Theorem 6.2. *Consider the DP problem $(\mathcal{P}, \mathcal{R})$. Let π be a simple projection such that $\mathcal{P} \subseteq \trianglerighteq_\pi$. Then, the function returning $\{(\mathcal{P} \setminus \rhd_\pi, \mathcal{R})\}$, is a sound DP processor.*

Here, for an arbitrary relation R, by R_π, we denote $\{(s,t) \mid (\pi(s), \pi(t)) \in R\}$. (In principle, there would be some additional syntactic restrictions. See for example Thiemann [42] and Sternagel and Thiemann [38]. For DP problems originating from TRSs, those are always satisfied. We will come back to this issue when we present our Isabelle formalization.)

The nice thing, when formulating the subterm criterion as a processor, is that we do not have to care about dependency graphs any more (there are separate processors specifically for incorporating the information provided by approximated dependency graphs).

6.3 Generalized Subterm Criterion

Only recently, a generalization of the subterm criterion (together with a Coq formalization) was given by Contejean et al. [7]. Before we can address the differences to the original subterm criterion, we need to be more formal about simple projections.

Definition 6.2 (Simple Projections). A *simple projection* for a set of function symbols $\mathcal{F}$, is a mapping π that assigns to every $(f, n) \in \mathcal{F}$ an argument position $0 < i \leq n$. This is extended to terms by $\pi(f(t_1, \ldots, t_n)) = t_i$.

The first generalization, is replacing simple projections by what is called *projections* in [7].

Definition 6.3 (Projections). A *projection* for a set of function symbols $\mathcal{F}$, is a mapping π that assigns to every $(f, n) \in \mathcal{F}$ an arbitrary natural number i. The extension to terms is defined by

$$\pi(t) = \begin{cases} t_i & \text{if } t = f(t_1, \ldots, t_n) \text{ and } 0 < i \leq n \\ t & \text{otherwise} \end{cases}$$

This means that when using projections instead of simple projections, we gain the possibility of identity mappings (that is, not projecting a term at all).

Now, the generalized subterm criterion was introduced as follows:

Theorem 6.3 (Contejean et al. [7], Theorem 4). *Let $\mathcal{R}$ be a TRS and $\mathcal{P}$ be a subset of $\mathsf{DP}(\mathcal{R})$. If there exists a projection π such that*

- *$\mathcal{P}$ can be split into $\mathcal{P}' \uplus (s, t)$,*
- *$t \rhd \pi(t)$,*
- *$\pi(s)$ $(\rhd \cup \to_{\mathcal{R}})^+$ $\pi(t)$,*
- *and for every pair $(u, v) \in \mathcal{P}'$ that is strongly connected to (s, t), we have $\pi(u)$ $(\rhd \cup \to_{\mathcal{R}})^*$ $\pi(v)$,*

then, finiteness of $(\mathcal{P}', \mathcal{R})$ implies finiteness of $(\mathcal{P}, \mathcal{R})$ (where we use a slightly different notation from that actually used in [7]).

We now see that the second generalization was to allow $(\rhd \cup \to_{\mathcal{R}})^+$ $((\rhd \cup \to_{\mathcal{R}})^*)$ instead of just $\rhd$ $(\trianglerighteq)$. The statement involving "strongly connected to," seems to be equivalent to incorporating the dependency graph in the definition of the generalized subterm criterion.

6.4 Generalized Subterm Criterion Processor

In [38], we gave a formalization of Theorem 6.2 that was obtained independently from the development of Contejean et al. [7]. After examining Theorem 6.3 and consulting our own Isabelle formalization, we found that within the soundness proof we actually did already use the more general relation (involving rewrite

steps) and it just did not occur to us that its usage could be lifted to the outermost level (which can be verified by browsing the freely available mercurial repository of our development). Switching from simple projections to projections did not pose any problems either. This leads us to the latest version of the generalized subterm criterion as formalized in IsaFoR.

Theorem 6.4 (Generalized Subterm Criterion Processor). *Consider the DP problem $(\mathcal{P}, \mathcal{R})$. Let π be a projection such that*

1. $\mathcal{P} - \mathcal{P}' \subseteq ((\rhd \cup \to_{\mathcal{R}})^+)_\pi$,
2. $\mathcal{P}' \subseteq =_\pi$, *and*
3. $\forall (s,t) \in \mathcal{P}.\, s \notin \mathcal{V} \wedge t \notin \mathcal{V} \wedge \mathsf{root}(t) \notin \mathcal{D}_{\mathcal{R}}$.

Then, the function returning $\{(\mathcal{P}', \mathcal{R})\}$, is a sound DP processor.

Note that condition 3 will always be satisfied by DP problems that stem from a TRS $\mathcal{R}$, due to the construction of $\mathsf{DP}(\mathcal{R})$. Hence, it seems as if we have an even more general criterion than the one from Theorem 6.3, since we do not require that for right-hand sides of removed rules the projection was actually a simple one (we will put this claim into perspective in the last section).

In the next section we give the soundness proof of the generalized subterm criterion processor, as formalized in IsaFoR.

6.4.1 Soundness

We prove soundness of the generalized subterm criterion processor by showing that it preserves minimal infinite chains. In fact, we even show a slightly stronger result, namely that the subterm criterion processor is *chain identifying*. For shifting an infinite sequence f by j positions to the left, we use the function

$$\mathit{shift}\ f\ j = (\lambda i.\ f\ (i + j))$$

Definition 6.4 (Chain-Identifying DP Problems). Essentially, a DP problem $(\mathcal{P}', \mathcal{R}')$ is said to be *chain-identifying* with respect to $(\mathcal{P}, \mathcal{R})$, whenever every minimal infinite chain in $(\mathcal{P}, \mathcal{R})$ has a tail in $(\mathcal{P}', \mathcal{R}')$. The formal definition is:

$\mathsf{chain\text{-}id}\ (\mathcal{P}, \mathcal{R})\ (\mathcal{P}', \mathcal{R}') \equiv$

$\mathcal{R}' \subseteq \mathcal{R} \wedge$

$(\forall\, s\ t\ \sigma.$

$\quad \mathsf{ichain}_{\mathsf{min}}\ (\mathcal{P},\ \mathcal{R})\ s\ t\ \sigma \longrightarrow$

$\quad (\exists\, i.\ \mathsf{ichain}\ (\mathcal{P}',\ \mathcal{R}')\ (\mathit{shift}\ s\ i)\ (\mathit{shift}\ t\ i)\ (\mathit{shift}\ \sigma\ i)))$

Now, instead of just saying that for every minimal infinite chain in the original DP problem, there exists a minimal infinite chain in the new DP problem (as is done in the case of soundness), we get a direct connection between those two chains: after a finite number of steps, the minimal chain in the original DP problem is *identical* to the chain in the new DP problem. Together with the condition $\mathcal{R}' \subseteq \mathcal{R}$, this implies that the newly obtained chain is also minimal. Hence, chain-identifyingness impliess soundness (as has been shown in IsaFoR by the lemma *ci_proc_to_sound_proc*).

Since, the subterm criterion processor does not modify $\mathcal{R}$, the first conjunct trivially holds. The advantage of showing chain-identifyingness over just preservation of minimal infinite chains, is that in the presence of labeling transformations (see also Chapter 8), it is allowed to *remove* labels, whenever only chain-identifying processors have been used before. We do not want to go into details here. Just remember, that chain-identifyingness implies soundness of processors.

Recall our assumptions:

assumes *1*: $\mathcal{P} - \mathcal{P}' \subseteq ((\rhd \cup \to_{\mathcal{R}})^+)_\pi$
and *2*: $\mathcal{P}' \subseteq =_\pi$
and *3*: $\forall (s,\ t) \in \mathcal{P}.\ s \notin \mathcal{V} \wedge t \notin \mathcal{V} \wedge (\mathsf{root}(t),\ \mathit{num\_args}\ t) \notin \mathcal{D}_{\mathcal{R}}$
and *mchain*: $\mathsf{ichain}_{\mathsf{min}}\ (\mathcal{P},\ \mathcal{R})\ s\ t\ \sigma$
shows $\exists\, i.\ \mathsf{ichain}\ (\mathcal{P}',\ \mathcal{R})\ (\mathit{shift}\ s\ i)\ (\mathit{shift}\ t\ i)\ (\mathit{shift}\ \sigma\ i)$
proof −

We proceed by a case distinction: Either there is a point, from which on only pairs of $\mathcal{P}'$ are used, or not.

have $(\exists\, j.\ \forall\, i \geq j.\ (s\ i,\ t\ i) \in \mathcal{P}') \vee (\forall\, i.\ \exists\, j \geq i.\ (s\ j,\ t\ j) \notin \mathcal{P}')$ **by** *auto*
thus *?thesis*

proof

Assume that there is an index j such that all later pairs are in $\mathcal{P}'$. Then, we can construct an infinite chain, by just shifting all indices in the original sequences by j.

assume $\exists j.\ \forall i{\geq}j.\ (s\ i,\ t\ i) \in \mathcal{P}'$
then obtain j **where** *tail*: $\forall i{\geq}j.\ (s\ i,\ t\ i) \in \mathcal{P}'$ **by** *auto*
let *?s* = *shift s j* **and** *?t* = *shift t j* **and** *?σ* = *shift σ j*
from *mchain* **have** *rsteps*: $\forall i.\ (t\ i)(\sigma\ i) \rightarrow_{\mathcal{R}}^{*} (s\ (i{+}1))(\sigma\ (i{+}1))$ **by** *simp*
from *tail* **have** $\forall i.\ (?s\ i,\ ?t\ i) \in \mathcal{P}'$ **by** *auto*
moreover from *rsteps*
have $\forall i.\ (?t\ i)(?\sigma\ i) \rightarrow_{\mathcal{R}}^{*} (?s\ (i+1))(?\sigma\ (i+1))$ **by** *auto*
ultimately show *?thesis* **by** *auto*
next

In the other case, we will always find an index where a pair from $\mathcal{P}$ is used, independently from how far we proceed inside our infinite chain.

assume $\forall i.\ \exists j{\geq}i.\ (s\ j,\ t\ j) \notin \mathcal{P}'$

Once again, the Axiom of Choice comes in handy by providing a function f giving those indices. That is, for every i, we get an index $f\ i$ which is greater than or equal to i, such that the $f\ i$-th pair of the sequence is not in $\mathcal{P}'$.

from *choice*[*OF this*] **obtain** f::*nat* ⇒ *nat* **where** *f_geq*: $\forall i.\ f\ i \geq i$
and *P_seq*: $\forall i.\ (s\ (f\ i),\ t\ (f\ i)) \in \mathcal{P} - \mathcal{P}'$ **using** *mchain* **by** *auto*

From assumption *3*, we get that all the $s\ i$ and $t\ i$ are non-variable terms. Further, since applying a substitution to a function application does not change the root symbol, the root of $(t\ i)(\sigma\ i)$ is not defined in $\mathcal{R}$.

from *3* **and** *mchain*
have *s_no_vars*: $\forall i.\ (s\ i) \notin \mathcal{V}$ **and** *t_no_vars*: $\forall i.\ (t\ i) \notin \mathcal{V}$ **by** *auto*
have *undef_root*: $\forall i.\ (\mathsf{root}((t\ i)(\sigma\ i)), num\_args\ ((t\ i)(\sigma\ i))) \notin \mathcal{D}_{\mathcal{R}}$ ⟨*proof omitted*⟩

To shorten the following statements we introduce some abbreviations. We use *?S* instead of $\rhd \cup \rightarrow_{\mathcal{R}}$ and *?s* as well as *?t* for the projected sequences with applied substitutions, respectively.

let *?S* = $\rhd \cup \rightarrow_{\mathcal{R}}$
let *?s* = $\lambda i.\ \pi\ ((s\ i)(\sigma\ i))$
let *?t* = $\lambda i.\ \pi\ ((t\ i)(\sigma\ i))$

Now, we show that all the $\pi\ (s\ i)(\sigma\ i)$ are connected by $\rhd \cup \rightarrow_{\mathcal{R}}$-sequences.

have *seq*: $\forall\, i.\ (?s\ i,\ ?s\ (i+1)) \in ?S^*$
proof
fix *i*

For every single pair $(s\ i,\ t\ i)$, we know that it is either in $\mathcal{P}$ without $\mathcal{P}'$, or that it is in $\mathcal{P}'$ itself. This fact is used in a case distinction to show that the projected pairs (with applied substitutions) are connected by $(\rhd \cup \rightarrow_{\mathcal{R}})^*$.

from *mchain* **have** $(s\ i,\ t\ i) \in (\mathcal{P} - \mathcal{P}') \cup \mathcal{P}'$ **by** *simp*
hence $(?s\ i,\ ?t\ i) \in ?S^*$
proof

In the former case, we obtain the desired result with the help of condition *1*, together with the fact

supt_rstep_trancl_stable:
$\bigwedge s\ t\ \mathcal{R}\ \sigma.\ (s,\ t) \in (\rhd \cup \rightarrow_{\mathcal{R}})^+ \Longrightarrow (s\sigma,\ t\sigma) \in (\rhd \cup \rightarrow_{\mathcal{R}})^+$

assume $(s\ i,\ t\ i) \in \mathcal{P} - \mathcal{P}'$
with *1* **have** $(\pi\ (s\ i),\ \pi\ (t\ i)) \in ?S^+$ **by** *auto*
from *supt_rstep_trancl_stable*[*OF this*, *of* $\sigma\ i$]
have $(?s\ i,\ ?t\ i) \in ?S^+$
using *s_no_vars*[*THEN spec*, *of i*]
and *t_no_vars*[*THEN spec*, *of i*] **by** *simp*
thus *?thesis* **..**
next

In the latter case, condition *2* suffices.

assume $(s\ i,\ t\ i) \in \mathcal{P}'$
with *2* **have** $\pi\ (s\ i) = \pi\ (t\ i)$ **by** *auto*
hence $(\pi\ (s\ i))(\sigma\ i) = (\pi\ (t\ i))(\sigma\ i)$ **by** *simp*
hence $?s\ i = ?t\ i$
using *s_no_vars*[*THEN spec*, *of i*]

```
    and t_no_vars[THEN spec, of i] by simp
  thus ?thesis by simp
qed
```

Having the part from $\pi\ (s\ i)(\sigma\ i)$ to $\pi\ (t\ i)(\sigma\ i)$, we still have to show that $(\rhd \cup \to_{\mathcal{R}})^*$ leads us from $\pi\ (t\ i)(\sigma\ i)$ to $\pi\ (s\ (i{+}1))(\sigma\ (i{+}1))$. Here, we use the fact that a rewrite sequence starting at a terminating term whose root is not defined, is preserved by applying a projection. A proof of *SN_elt_rsteps_proj_term* can be found in Appendix A. Further, *rtrancl_Un_subset* is part of Isabelle's library and provides the fact:

rtrancl_Un_subset: $R^* \cup S^* \subseteq (R \cup S)^*$

```
  moreover have (?t i, ?s (i + 1)) ∈ ?S*
  proof −
    from mchain have sn: SN_R((t i)(σ i)) by (simp)
    from mchain have (t i)(σ i) →*_R (s (i + 1))(σ (i + 1)) by simp
    from SN_elt_rsteps_proj_term[OF this sn undef_root[THEN spec, of i]]
      show ?thesis using rtrancl_Un_subset[of ▷ →_R] by auto
  qed
```

Combining the above two facts yields the desired result.

```
  ultimately show (?s i, ?s (i + 1)) ∈ ?S* by simp
qed
```

Since all the $\pi\ (s\ i)(\sigma\ i)$ are connected by $(\rhd \cup \to_{\mathcal{R}})^*$, we can reach every larger index, starting from $\pi\ (s\ i)(\sigma\ i)$, using $\rhd \cup \to_{\mathcal{R}}$.

```
from iseq_imp_steps[OF seq] and f_geq
  have between: ∀ i. (?s i, ?s (f i)) ∈ ?S* by simp
```

Now, we use the same trick as already for *non_strict_ending*, by using the function *shift_by* in order to pinpoint those indices of the sequence that we obtain by the choice function *f*. We abbreviate the resulting sequence by *?s′*.

```
let ?s′ = λi. π (s (shift_by f (i + 1)))(σ (shift_by f (i + 1)))
```

Before we can show that the *?s′ i* are connected by non-empty *?S* sequences, we need to get a connection between *?s i* and *?s* $((f\ i) + 1)$.

have $\forall\, i.\ (?s\ i,\ ?s\ ((f\ i)+1)) \in ?S^+$
proof
fix i
from *P_seq* **and** *1* **have** $(\pi\ (s\ (f\ i)),\ \pi\ (t\ (f\ i))) \in ?S^+$ **by** *auto*
from *supt_rstep_trancl_stable*[*OF this*]
have $(\pi\ ((s\ (f\ i))(\sigma\ (f\ i))),\ \pi\ ((t\ (f\ i))(\sigma\ (f\ i)))) \in ?S^+$
using *s_no_vars*[*THEN spec*, *of f i*] *t_no_vars*[*THEN spec*, *of f i*]
by *simp*
with *between*[*THEN spec*, *of i*]
have $(?s\ i,\ \pi\ ((t\ (f\ i))(\sigma\ (f\ i)))) \in ?S^+$ **by** *simp*
moreover
have $(\pi\ ((t\ (f\ i))(\sigma\ (f\ i))),\ \pi\ ((s\ ((f\ i)+1))(\sigma\ ((f\ i)+1)))) \in ?S^*$
proof −
from *mchain*
have *sn*: $\mathsf{SN}_{\mathcal{R}}((t\ (f\ i))(\sigma\ (f\ i)))$ **by** (*simp*)
from *mchain*
have $(t\ (f\ i))(\sigma\ (f\ i)) \to^*_{\mathcal{R}} (s\ ((f\ i)+1))(\sigma\ ((f\ i)+1))$ **by** *simp*
from *SN_elt_rsteps_proj_term*[*OF this sn*
undef_root[*THEN spec*, *of f i*], *of* π]
show *?thesis*
using *rtrancl_Un_subset*[*of* $\rhd \to_{\mathcal{R}}$] **by** *auto*
qed
ultimately show $(?s\ i,\ ?s\ ((f\ i)+1)) \in ?S^+$ **by** *simp*
qed
hence $\forall\, i.\ (?s'\ i,\ ?s'\ (i+1)) \in ?S^+$ **by** *simp*
with *refl*[*of* $?s'\ 0$]
have $\exists\, S.\ S\ 0 = ?s'\ 0 \wedge (\forall\, i.\ (S\ i,\ S\ (i+1)) \in ?S^+)$ **by** *best*

Thus we have constructed an infinite $?S^+$-sequence, starting at $?s'\ 0$.

hence $\neg\ \mathsf{SN}_{?S^+}(?s'\ 0)$ **unfolding** *SN_defs* **by** *simp*

But $?s'\ 0$ is terminating with respect to $?S^+$, yielding a contradiction.

moreover have $\mathsf{SN}_{?S^+}(?s'\ 0)$
proof −

Proving that $?s'\ 0$ is terminating with respect to $?S^+$, proceeds as follows.

First we show that we reach $?s'\ 0$ from the initial term $\pi\ (t\ 0)(\sigma\ 0)$, by using $?S$.

```
have (?t 0, ?s' 0) ∈ (?S⁺)*
proof −
  from mchain have sn: SN_R((t 0)(σ 0)) by (simp)
  from mchain have (t 0)(σ 0) →*_R (s (0 + 1))(σ (0 + 1)) by simp
  from SN_elt_rsteps_proj_term[OF this sn
    undef_root[THEN spec, of 0], of π]
    have (?t 0, ?s (0 + 1)) ∈ ?S*
    using rtrancl_Un_subset[of ▷ →_R] by auto
  moreover have (?s (0 + 1), ?s' 0) ∈ ?S*
  proof −
    have shift_by f (0 + 1) ≥ (0 + 1) by simp
    from iseq_imp_steps[OF seq this] show ?thesis by simp
  qed
  ultimately show ?thesis by simp
qed
```

Then, we show that $\pi\ (t\ 0)(\sigma\ 0)$ does not allow infinite $?S^+$-sequences. Here, we use the fact that termination with respect to $\rightarrow_{\mathcal{R}}$, implies termination with respect to $\triangleright \cup \rightarrow_{\mathcal{R}}$. The proof of *SN_elt_rstep_imp_SN_elt_supt_union_rstep* is to be found in Appendix A.

```
moreover have SN_{?S⁺}(?t 0)
proof −
  from mchain have SN_R((t 0)(σ 0)) by (simp)
  moreover have (t 0)(σ 0) ⊵ ?t 0 by (simp add: supteq_proj_term)
  ultimately have SN_R(?t 0)
    using subterm_preserves_SN[of R (t 0)(σ 0) ?t 0] by auto
  from SN_elt_imp_SN_elt_trancl[OF
    SN_elt_rstep_imp_SN_elt_supt_union_rstep[OF this]]
    show ?thesis .
qed
```

Then, since descendants of terminating terms are also terminating, we get that $?s'\ 0$ is terminating.

```
ultimately show ?thesis by (rule steps_preserve_SN_elt)
```

```
    qed
    ultimately show ?thesis ..
  qed
qed
```

6.5 Practicability

As usual, there is not just one viewpoint. We do at last have the perspective of certifiers (taking full proofs and 'merely' checking their correctness) as well as the perspective of termination tools (trying to find as many proofs as possible, as fast as possible).

For a certifier, the generalized subterm criterion (processor) is clearly an advancement over the old version. But how about termination tools? Do they have to change their implementations of the subterm criterion in order to accommodate identity mappings and the possibility of rewrite steps? In the following we will argue that this is not the case.

6.5.1 Identity Mappings

Let us first make some (commonly accepted) restrictions. In the remainder, whenever we talk about a DP problem $(\mathcal{P}, \mathcal{R})$, we implicitly assume that for all $(s, t) \in \mathcal{P}$, s and t are not variables. We further assume that the root symbols of s and t do not occur anywhere below the root in $\mathcal{P}$, and not at all in $\mathcal{R}$. We denote such function symbols by capital letters, that is, $s = F(s_1, \ldots, s_m)$ and $t = G(t_1, \ldots, t_n)$. This assumptions are naturally satisfied by the construction of the dependency pairs (for well-formed TRSs). Further, virtually every existing DP processor preserves this assumptions.

Now consider some DP problem $(\mathcal{P}, \mathcal{R})$ where the generalized subterm criterion processor is applicable.

We will first show that it is not possible for an identity mapping to occur just at the right-hand side of a pair $(s, t) \in \mathcal{P}$. Assume to the contrary that it does, that is, there is a pair $(s, t) \in \mathcal{P}$, such that $s = F(s_1, \ldots s_m)$ with $\pi(F) = i \in \{1, \ldots, m\}$ and $t = G(t_1, \ldots, t_n)$ with $\pi(G) = j \notin \{1, \ldots, n\}$.

Then, $\pi(s) = s_i$ and $\pi(t) = G(\ldots)$. But the symbol G may neither occur in s_i (since it is below the root of a left-hand side of $\mathcal{P}$), nor in any rule of $\mathcal{R}$. Thus, no rewrite sequence starting from some subterm of s_i could ever introduce G. This implies that $\pi(s)\ (\rhd \cup \to_{\mathcal{R}})^*\ \pi(t)$ is impossible and hence contradicts the applicability of the generalized subterm criterion.

Next, we show that we can also get rid of the cases where an identity mapping does just occur at the left-hand side of some pair. Assume that we have such a pair, that is, $(s,t) \in \mathcal{P}$ where $s = F(s_1,\ldots,s_m)$ with $\pi(F) = i \notin \{1,\ldots,m\}$ and $t = G(t_1,\ldots,t_n)$ with $\pi(G) = j \in \{1,\ldots,n\}$. This implies $F \neq G$. We further know that F (since it uses an identity mapping) does not occur just on the right-hand side of any pair in $\mathcal{P}$. But then, (s,t) cannot be strongly connected to any other pair in $\mathcal{P}$ (since once we leave the set of pairs having F as root of both sides by means of (s,t) there is no way to come back). For Theorem 6.3, if (s,t) is the pair to be removed, we can just take an arbitrary projection to some s_i and will still be able to apply the criterion (since no other pair of $\mathcal{P}$ is strongly connected to (s,u)); otherwise (for the same reason) we can ignore the pair (s,t) completely. Concerning Theorem 6.4, it is enough to assume that our supposed termination tool actually generating the proof, uses at least the weakest form of dependency graph analysis, since then we will only have to consider strongly connected components of the dependency graph, but (s,t) can never be part of such components.

Finally, consider that there is some pair $(s,t) \in \mathcal{P}$ where on both sides an identity mapping is used. That is, $s = F(s_1,\ldots,s_m)$ with $\pi(F) \notin \{1,\ldots,m\}$ and $t = G(t_1,\ldots,t_n)$ with $\pi(G) \notin \{1,\ldots,n\}$. Considering $\pi(s)\ (\rhd \cup \to_{\mathcal{R}})^*\ \pi(t)$, we can restrict to $\pi(s) \to_{\mathcal{R}}^*\ \pi(t)$, since G may not occur below the root of s. Further, rewriting cannot change the root of s (since it does not occur in $\mathcal{R}$) and hence $F = G$ and $m = n$. But then $\pi(s) \to_{\mathcal{R}}^*\ \pi(t)$ implies that for all $k \in \{1,\ldots,n\}$, we have $s_k \to_{\mathcal{R}}^*\ t_k$. For Theorem 6.3 this is already enough (since we do only remove one pair at a time and can thus just take the appropriate k for this rule). Concerning Theorem 6.4, it could be the case that there where several pairs of the form $(F(\ldots),F(\ldots))$ that could be removed all at once. Now only those can be removed for which there is a non-empty rewrite sequence in the k-th argument (note that for all the others the k-th argument stays the same). This is not really a problem, since we can take almost the same simple projection where we just change the k appropriately for a further

application of the processor. So, the worst that could happen, is that we now need several applications of Theorem 6.4 instead of just one.

Example 6.2. Consider the following (contrived) TRS for addition and multiplication.

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

Of the five dependency pairs, only two remain, after decomposing the estimated dependency graph and applying the original version of the subterm criterion, where we use M instead of $\times^\sharp$:

$$\begin{aligned} \mathsf{M}(\mathsf{s}(x) + y, z) &\to \mathsf{M}(\mathsf{s}(x + y), z) \\ \mathsf{M}(x, \mathsf{s}(y) + z) &\to \mathsf{M}(x, \mathsf{s}(y + z)) \end{aligned}$$

Using an identity mapping for M we can get rid of both DPs at once (by a single rewrite step, respectively).

Now, if we ban identity projections, we have to fix an argument position. Regardless of the position we choose, we can remove one DP (since there is a rewrite step) and remain with the other (where the projected left- and right-hand sides are equal). By a second application of the generalized subterm criterion without identity mappings, this time choosing a different position than before, we get rid of the remaining dependency pair. (Note that this example cannot be proven terminating by just using the original version of the subterm criterion together with the decomposition of the estimated dependency graph.)

This indicates that identity mappings are not useful in practice and should thus be avoided by termination tools, since taking them into account would just increase the search space.

6.5.2 Rewrite Steps

Recall that what makes the subterm criterion so attractive is its speed and its simplicity. When expanding an existing implementation of the subterm

criterion to take also rewriting steps into account, this will have a negative impact on its performance. Even worse, up to date, there is no automatically found proof using the generalized subterm criterion, which could not also be found by just using the subterm criterion in its original version. This seems to indicate that termination tools do not really have to worry about changing their implementations of the subterm criterion.

6.6 Chapter Notes

We have shown how the subterm criterion [17, 18] evolved from its original version to a (generalized) processor of the DP framework [7, 38]. We further indicated that the latest advancements for the subterm criterion, although nice in theory, do not have a great impact on termination tools. Inside IsaFoR, our formalization of the subterm criterion is combined with a formalization of size-change termination due to Krauss [24] to yield a size-change processor based on the subterm relation.

Chapter 7

Signature Extensions

A signature always reveals a man's character—and sometimes even his name.

Evan Esar

So, how was your weekend?
Did you spend it chasing down wild signatures?

Billy
Dr. Horrible's Sing-Along Blog

In the literature, terms are usually built with respect to some *signature* and some countably infinite set of *variables*. Here, a signature is a set of function symbol arity pairs, where the *arity* of a function symbol determines, how many arguments the function symbol takes. However, neither of those restrictions applies to our (α, β) *term* data type. In this chapter, we go into restricting terms to a given signature. For termination analysis, it is important to know whether termination results for terms that are built exclusively over the function symbols occurring in a given TRS, extend to arbitrary terms. Put differently: after proving termination of a given TRS, can we be sure that the result holds for arbitrary inputs?

7.1 Well-Formed Terms

Our first goal, is to show that it suffices to prove termination for terms built over $\mathcal{F}(\mathcal{R})$ (see Definition 4.6), in order to conclude termination of $\mathcal{R}$ for arbitrary terms. Therefor, we need the inductively defined set of well-formed terms.

Definition 7.1 (Well-Formed Terms). Given a signature $\mathcal{F}$, the set of *well-formed terms* with respect to $\mathcal{F}$, is defined by the rules:

$$\frac{}{Var\ x \in \mathcal{T}(\mathcal{F})} \qquad \frac{(f, |ts|) \in \mathcal{F} \quad \forall\, t{\in}ts.\ t \in \mathcal{T}(\mathcal{F})}{Fun\ f\ ts \in \mathcal{T}(\mathcal{R})}$$

Hence, a variable is a well-formed term and, whenever all elements of the list ts are well-formed terms and additionally f with arity $|ts|$ is in $\mathcal{F}$, we may conclude that the composed term $Fun\ f\ ts$ is well-formed.

Reasoning about Well-Formed Terms.
The well-formed terms over a signature $\mathcal{F}$ are formalized by the inductively defined set $\mathcal{T}(\mathcal{F})$ (internally, we use the name *wf_terms*), constructed by the rules:

wf_terms.Var: $Var\ x \in \mathcal{T}(\mathcal{F})$
wf_terms.Fun: $[\![(f, |ts|) \in \mathcal{F};\ \bigwedge t.\ t \in ts \Longrightarrow t \in \mathcal{T}(\mathcal{F})]\!] \Longrightarrow Fun\ f\ ts \in \mathcal{T}(\mathcal{F})$

Definition 7.2 (Well-Formed Contexts). Further, the set of *well-formed contexts* with respect to $\mathcal{F}$, is given by:

$$\frac{}{\Box \in \mathcal{C}(\mathcal{F})} \qquad \frac{(f, |ss\ @\ ts| + 1) \in \mathcal{F} \quad \forall\, t{\in}ss\ @\ ts.\ t \in \mathcal{T}(\mathcal{F}) \quad D \in \mathcal{C}(\mathcal{F})}{More\ f\ ss\ D\ ts \in \mathcal{C}(\mathcal{F})}$$

Clearly, the empty context is well-formed (since otherwise, the set of well-formed contexts would be empty). We obtain a well-formed context from a function symbol f, two lists of well-formed terms ss and ts, and a well-formed context D, whenever the number of terms in ss and ts, together with D, corresponds to the arity of f.

Reasoning about Well-Formed Contexts.
Again, we use an inductively defined set. The internal name of $\mathcal{C}(\mathcal{F})$ is *wf_ctxts* $\mathcal{F}$ and it is defined by the rules:

wf_ctxts.Hole: $\Box \in \mathcal{C}(\mathcal{F})$
wf_ctxts.More:
$[\![(f, |ss\ @\ ts| + 1) \in \mathcal{F};\ \bigwedge t.\ t \in ss\ @\ ts \Longrightarrow t \in \mathcal{T}(\mathcal{F});\ D \in \mathcal{C}(\mathcal{F})]\!]$
$\Longrightarrow More\ f\ ss\ D\ ts \in \mathcal{C}(\mathcal{F})$

Now, proving termination for terms built over $\mathcal{F}(\mathcal{R})$, is the same as proving termination for the rewrite relation $\rightarrow_{\mathcal{R}}$, but restricted to well-formed terms over $\mathcal{F}(\mathcal{R})$.

Definition 7.3 (Well-Formed Rewrite Relation). The *well-formed rewrite relation* induced by a TRS $\mathcal{R}$, is defined as follows:

$$\twoheadrightarrow_{\mathcal{R}} \equiv \{(s,\, t) \mid s \to_{\mathcal{R}} t \wedge s \in \mathcal{T}(\mathcal{F}(\mathcal{R})) \wedge t \in \mathcal{T}(\mathcal{F}(\mathcal{R}))\}$$

Reasoning about the well-formed Rewrite Relation.
The ASCII name of $\twoheadrightarrow_{\mathcal{R}}$ is *wfrstep* $\mathcal{R}$. Hence, its definition is available under the name *wfrstep_def*.

7.2 Signature Extensions Preserve Termination

Having the well-formed rewrite relation, we can state the theorem about termination preservation as follows.

Theorem 7.1. $\mathsf{SN}(\twoheadrightarrow_{\mathcal{R}}) = \mathsf{SN}(\mathcal{R})$

Since the direction from right to left is trivial, we concentrate on the converse (actually in its contrapositive form): $\neg\ \mathsf{SN}(\mathcal{R}) \Longrightarrow \neg\ \mathsf{SN}(\twoheadrightarrow_{\mathcal{R}})$. First, consider the auxiliary function called *clean*

$$[\![Var\ y]\!]_{\mathcal{F}} = Var\ y$$
$$[\![Fun\ f\ ts]\!]_{\mathcal{F}} = (\mathit{if}\ (f,\, |ts|) \in \mathcal{F}\ \mathit{then}\ Fun\ f\ (map\ [\![\cdot]\!]_{\mathcal{F}}\ ts)\ \mathit{else}\ z)$$

which is used to remove unwanted function symbols from terms. Here, z is an arbitrary but fixed variable. That is, $[\![\cdot]\!]_{\mathcal{F}}$ replaces every maximal subterm of its argument that has a root which is not in $\mathcal{F}$, by z. The general structure of our proof consists of the following three parts:

1. First, we assume that $\mathsf{WF}(\mathcal{R})$ and $\neg\ \mathsf{SN}(\mathcal{R})$. Then, by Theorem 5.1, we obtain an infinite $(\mathsf{DP}(\mathcal{R}), \mathsf{ID}(\mathcal{R}))$-chain.
2. Then, we show that every infinite chain can be transformed into a *clean* infinite chain.
3. Finally, we show completeness of the DP transformation for well-formed terms, that is, a clean infinite $(\mathsf{DP}(\mathcal{R}), \mathsf{ID}(\mathcal{R}))$-chain can be transformed into an infinite $\twoheadrightarrow_{\mathcal{R}}$-rewrite sequence.

Putting all this together, we obtain $[\![\mathsf{WF}(\mathcal{R}); \mathsf{SN}(\twoheadrightarrow_{\mathcal{R}})]\!] \Longrightarrow \mathsf{SN}(\mathcal{R})$. Since every non well-formed TRS is nonterminating, this results in a proof of Theorem 7.1.

It remains to show the following two lemmas.

$[\![\mathcal{F}\mathsf{un}(\mathcal{P},\,\mathcal{R}) \subseteq \mathcal{F};\ \mathsf{ichain}\ (\mathcal{P},\,\mathcal{R})\ s\ t\ \sigma]\!] \Longrightarrow \mathsf{ichain}\ (\mathcal{P},\,\mathcal{R})\ s\ t\ [\![\sigma]\!]_{\mathcal{F}}$
$\mathsf{ichain}\ (\mathsf{DP}(\mathcal{R}),\,\mathsf{ID}(\mathcal{R}))\ s\ t\ [\![\sigma]\!]_{\sharp(\mathcal{F}(\mathcal{R}))} \Longrightarrow \neg\ \mathsf{SN}(\twoheadrightarrow_{\mathcal{R}})$

where we use the abbreviations $\mathcal{F}\mathsf{un}(\mathcal{P},\,\mathcal{R}) \equiv \mathcal{F}(\mathcal{P}) \cup \mathcal{F}(\mathcal{R})$ and $\sharp(\mathcal{R}) \equiv \{(\mathit{Plain}\ f,\ n) \mid (f,\ n) \in \mathcal{R}\} \cup \{(\mathit{Sharp}\ f,\ n) \mid (f,\ n) \in \mathcal{R}\}$.

7.2.1 Cleaning Preserves Infinite Chains

The first lemma states that whenever there is an infinite chain, there is also an infinite chain where only 'clean' terms are used. Here, by definition, the sequences s and t consist solely of 'clean' terms and thus only the substitutions of σ need to be cleaned.

Lemma 7.1. *If there is an infinite $(\mathcal{P},\,\mathcal{R})$-chain over the sequences s, t, and σ, then cleaning σ (with respect to a superset of $\mathcal{F}\mathsf{un}(\mathcal{P},\,\mathcal{R})$) results in an infinite chain of terms that are well-formed (since the terms of s and t are already well-formed).*

lemma *ichain_imp_clean_ichain*:
  assumes $\mathcal{F}\mathsf{un}(\mathcal{P},\,\mathcal{R}) \subseteq \mathcal{F}$ **and** $\mathsf{ichain}\ (\mathcal{P},\,\mathcal{R})\ s\ t\ \sigma$
  shows $\mathsf{ichain}\ (\mathcal{P},\,\mathcal{R})\ s\ t\ [\![\sigma]\!]_{\mathcal{F}}$
proof −

As usual, we deconstruct an infinite chain into steps in $\mathcal{P}$ and sequences in $\mathcal{R}$.

  from ‹$\mathsf{ichain}\ (\mathcal{P},\,\mathcal{R})\ s\ t\ \sigma$› **have** *P_seq*: $\forall\, i.\ (s\ i,\ t\ i) \in \mathcal{P}$
    and *seq*: $\forall\, i.\ (t\ i)(\sigma\ i) \rightarrow_{\mathcal{R}}^{*} (s\ (i+1))(\sigma\ (i+1))$ **by** *auto*

By definition of $\mathcal{F}\mathsf{un}(\mathcal{P},\,\mathcal{R})$, we further know that the function symbols of $\mathcal{P}$ as well as those of $\mathcal{R}$, are contained in $\mathcal{F}$.

  from ‹$\mathcal{F}\mathsf{un}(\mathcal{P},\,\mathcal{R}) \subseteq \mathcal{F}$› **have** $\mathcal{F}(\mathcal{P}) \subseteq \mathcal{F}$ **and** $\mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}$ **by** *simp+*

This implies that all the $s\ i$ and $t\ i$ are well-formed with respect to $\mathcal{F}$.

  have *wf_s*: $\forall\, i.\ s\ i \in \mathcal{T}(\mathcal{F})$ ⟨*proof omitted*⟩
  have *wf_t*: $\forall\, i.\ t\ i \in \mathcal{T}(\mathcal{F})$ ⟨*proof omitted*⟩

Then, we use the fact that $\mathcal{R}$-sequences are preserved by cleaning (this is indeed the key lemma of this proof). The proof of this fact is to be found in the Appendix.

```
have clean_seq: ∀ i. [[(t i)(σ i)]]_F →*_R [[(s (i + 1))(σ (i + 1))]]_F
  using seq rsteps_imp_clean_rsteps[OF ‹F(R) ⊆ F›] by blast
```

It remains to 'distribute' the cleaning function over the application of substitutions.

```
have ∀ i. (t i)[[σ i]]_F →*_R (s (i + 1))[[σ (i + 1)]]_F
proof
  fix i
  have clean_t: [[(t i)(σ i)]]_F = (t i)[[σ i]]_F
    using clean_subst_apply_term[OF wf_t[THEN spec[of _ i]]]
    unfolding o_def by (cases σ i) auto
  have clean_s: [[(s (i + 1))(σ (i + 1))]]_F = (s (i + 1))[[σ (i + 1)]]_F
    using clean_subst_apply_term[OF wf_s[THEN spec[of _ i + 1]]]
    unfolding o_def by (cases σ (i + 1)) auto
  from clean_seq[THEN spec[of _ i]]
    show (t i)[[σ i]]_F →*_R (s (i + 1))[[σ (i + 1)]]_F
    unfolding clean_s clean_t .
qed
with P_seq show ?thesis by auto
qed
```

7.2.2 DP Transformation for Well-Formed Terms is Complete

The second lemma is a completeness result for the DP Transformation, when considering only well-formed terms. That is, whenever we have a clean infinite chain using the initial DP problem, then this implies nontermination of the corresponding well-formed rewrite relation.

Lemma 7.2. *If there is an infinite* $(\mathsf{DP}(\mathcal{R}), \mathsf{ID}(\mathcal{R}))$*-chain over well-formed terms (with respect to* $\sharp(\mathcal{F}(\mathcal{R}))$*), then the well-formed rewrite relation of* $\mathcal{R}$ *is not terminating.*

lemma *clean_ichain_imp_not_SN_wfrstep*:
assumes *ichain*: ichain (DP($\mathcal{R}$), ID($\mathcal{R}$)) $s\ t\ [\![\sigma]\!]_{\sharp(\mathcal{F}(\mathcal{R}))}$
shows $\neg$ SN($\Rightarrow\!\!\!\!\!\!\Rightarrow_{\mathcal{R}}$)

We start by a case distinction on whether $\mathcal{R}$ is well-formed or not. If it is not, then we trivially obtain nontermination of $\Rightarrow\!\!\!\!\!\!\Rightarrow_{\mathcal{R}}$ (using a construction very similar to the one of Lemma 4.1).

proof (*cases* WF($\mathcal{R}$))
case *False* **show** *?thesis* ⟨*proof omitted*⟩
next
case *True*
let $?\mathcal{F} = \sharp(\mathcal{F}(\mathcal{R}))$
let $?c = \lambda t.\ [\![t]\!]_{?\mathcal{F}}$
from *ichain* **have** *P_seq*: $\forall\, i.\ (s\ i,\ t\ i) \in$ DP($\mathcal{R}$)
and *seq*: $\forall\, i.\ (t\ i)(?c\ (\sigma\ i)) \rightarrow^*_{\mathsf{ID}(\mathcal{R})} (s\ (i+1))(?c\ (\sigma\ (i+1)))$
by *auto*

We define a function modifying a substitution such that its results are unsharped and cleaned. The function for unsharping is abbreviated to u, for brevity.

u (*Var* x) = *Var* x
u (*Fun* f ts) = *Fun* (*case* f *of Sharp* $g \Rightarrow g$ | *Plain* $g \Rightarrow g$) (*map* u ts)

let $?cu = \lambda\sigma.$ *case* σ *of Subst* $s \Rightarrow$ *Subst* $([\![\cdot]\!]_{\mathcal{F}(\mathcal{R})} \circ$ u $\circ\ s)$

By construction of the dependency pairs, for every $(u,\ v) \in$ DP($\mathcal{R}$), we obtain a context C, such that the unsharped version of u is a left-hand side of some rule in $\mathcal{R}$ and C[u v] is the corresponding right-hand side. Additionally, C is well-formed (since it is part of a rule).

have $\forall\, i.\ \exists\, C.\ C \in \mathcal{C}(\mathcal{F}(\mathcal{R})) \wedge$ (u $(s\ i)$, C[u $(t\ i)$]) $\in \mathcal{R}$
⟨*proof omitted*⟩

By the Axiom of Choice, we then obtain such a context for every $(s\ i,\ t\ i)$.

from *choice*[*OF this*] **obtain** C
where $\forall\, i.\ C\ i \in \mathcal{C}(\mathcal{F}(\mathcal{R})) \wedge$ (u $(s\ i)$, $(C\ i)$[u $(t\ i)$]) $\in \mathcal{R}$ **by** *best*
hence *wf_ctxt*: $\forall\, i.\ C\ i \in \mathcal{C}(\mathcal{F}(\mathcal{R}))$

and *rules*: $\forall\, i.\ (\mathsf{u}\ (s\ i),\ (C\ i)[\mathsf{u}\ (t\ i)]) \in \mathcal{R}$ **by** *auto*

By applying the (cleaned and unsharped) substitutions to the rules, we obtain a sequence of $\mathcal{R}$-rewrite steps taking place at the root (which are however, not necessarily connected to each other).

from *rules* **have** *root_steps*:
$\forall\, i.\ (\mathsf{u}\ (s\ i))(?cu\ (\sigma\ i)) \to_{\mathcal{R}} ((C\ i)(?cu\ (\sigma\ i)))[(\mathsf{u}\ (t\ i))(?cu\ (\sigma\ i))]$
using *rstep.id*[*THEN rstep.subst*, *of* _ _ $\mathcal{R}$] **by** *force*

For convenience we introduce some more abbreviations. Here, D is an auxiliary function that stacks contexts and applies substitutions as needed for reconstructing $\mathcal{R}$ steps from DP-steps. It is defined by

$$D\ C\ \sigma\ 0 = \Box$$
$$D\ C\ \sigma\ (i+1) = (D\ C\ \sigma\ i) \circ (C\ i)(\sigma\ i)$$

Hence $?C\ i$ denotes the context used in the i-th step, whereas $?s\ i$ and $?t\ i$ are the corresponding terms.

let $?C = D\ C\ (\lambda i.\ ?cu\ (\sigma\ i))$
let $?s = \lambda i.\ (?C\ i)[(\mathsf{u}\ (s\ i))(?cu\ (\sigma\ i))]$
let $?t = \lambda i.\ (?C\ (Suc\ i))[(\mathsf{u}\ (t\ i))(?cu\ (\sigma\ i))]$
have *ctxt_seq*: $\forall\, i.\ (?s\ i) \to_{\mathcal{R}} (?t\ i)$ **using** *root_steps* **by** *auto*

Since every rewrite step that is possible in $\mathsf{ID}(\mathcal{R})$, is also possible in $\mathcal{R}$, from $\forall\, i.\ (t\ i)(\llbracket\sigma\rrbracket_{\sharp(\mathcal{F}(\mathcal{R}))}\ i) \to^*_{\mathsf{ID}(\mathcal{R})} (s\ (i+1))(\llbracket\sigma\rrbracket_{\sharp(\mathcal{F}(\mathcal{R}))}\ (i+1))$, we obtain a corresponding rewrite sequence in $\mathcal{R}$.

from *seq*
have $\forall\, i.\ \mathsf{u}\ ((t\ i)(?c\ (\sigma\ i))) \to^*_{\mathcal{R}} \mathsf{u}\ ((s\ (i+1))(?c\ (\sigma\ (i+1))))$ (**is** $\forall\, i.\ ?Q\ i$)
using *rstepIDs_imp_rsteps* **by** *best*

This sequence can then be transformed by 'distributing' u over substitution-applications.

have $\forall\, i.\ (\mathsf{u}\ (t\ i))(?cu\ (\sigma\ i)) \to^*_{\mathcal{R}} (\mathsf{u}\ (s\ (i+1)))(?cu\ (\sigma\ (i+1)))$
(**is** $\forall\, i.\ ?P\ i$)
⟨*proof omitted*⟩

This yields the connection between the $?t\ i$ and $?s\ (i+1)$.

hence *steps*: $\forall\, i.\ (?t\ i) \to_{\mathcal{R}}^{*} (?s\ (i+1))$
using *rsteps_closed_ctxt* **by** *best*

In combination with the steps of *ctxt_seq*, we then obtain non-empty rewrite sequences between two consecutive elements of *?s*.

have *trancl*: $\forall\, i.\ (?s\ i) \to_{\mathcal{R}}^{+} (?s\ (i+1))$ ⟨*proof omitted*⟩

Further, all the *?C i* are well-formed, since they are parts of right-hand sides of $\mathcal{R}$.

have $\forall\, i.\ ?C\ i \in \mathcal{C}(\mathcal{F}(\mathcal{R}))$ **by** *simp*

Moreover, since every $s\ i$ is a left-hand side of some pair in $\mathsf{DP}(\mathcal{R})$ and *?cu* removes function symbols that are not in $\mathcal{F}(\mathcal{R})$, all the $(\mathsf{u}\ (s\ i))(?cu\ (\sigma\ i))$ are well-formed.

moreover have $\forall\, i.\ (\mathsf{u}\ (s\ i))(?cu\ (\sigma\ i)) \in \mathcal{T}(\mathcal{F}(\mathcal{R}))$ ⟨*proof omitted*⟩

Putting these two facts together, yields that the *?s i* are well-formed.

ultimately have $\forall\, i.\ ?s\ i \in \mathcal{T}(\mathcal{F}(\mathcal{R}))$ **by** *best*
with *trancl*
have *seq*: $\forall\, i.\ (?s\ i,\ ?s\ (i+1)) \in$
$\{(x,\ y) \mid x\ y.\ x \to_{\mathcal{R}}^{+} y \wedge x \in \mathcal{T}(\mathcal{F}(\mathcal{R})) \wedge y \in \mathcal{T}(\mathcal{F}(\mathcal{R}))\}$ (**is** $\forall\, i.\ \_ \in ?R$)
by *blast*

This enables us to build an infinite *?R*-sequence, starting at *?s 0* and thus proves nontermination of *?R*.

have *?s 0* = *?s 0* **..**
from *this* **and** *seq*
have $\exists\, S.\ S\ 0 = ?s\ 0 \wedge (\forall\, i.\ (S\ i,\ S\ (i+1)) \in ?R)$ **by** *best*
hence $\neg\ \mathsf{SN}(?R)$ **unfolding** *SN_defs* **by** *blast*

With the auxiliary lemma

wfrstep_trancl:
$\mathsf{WF}(\mathcal{R}) \Longrightarrow \twoheadrightarrow_{\mathcal{R}}^{+} = \{(x,\ y) \mid x \to_{\mathcal{R}}^{+} y \wedge x \in \mathcal{T}(\mathcal{F}(\mathcal{R})) \wedge y \in \mathcal{T}(\mathcal{F}(\mathcal{R}))\}$

we obtain nontermination of $\Rightarrow_{\mathcal{R}}^{+}$ and thus, using once more the fact that termination of a relation is equivalent to termination of its transitive closure, nontermination of $\Rightarrow_{\mathcal{R}}$.

hence $\neg$ SN($\Rightarrow_{\mathcal{R}}^{+}$) **unfolding** *wfrstep_trancl*[*OF True*] **.**
with *SN_trancl_SN_conv* **show** *?thesis* **by** *auto*
qed

7.2.3 Putting It All Together

First note that the previous two lemmas can also be used to obtain the classical completeness result of the dependency pair transformation.

Lemma 7.3 (Completeness of DP Transformation). *If the TRS $\mathcal{R}$ is terminating, then the initial DP problem is finite.*

lemma *DP_complete*:
assumes $\mathsf{ichain}_{\mathsf{min}}$ (DP($\mathcal{R}$), ID($\mathcal{R}$)) *s t* σ **shows** $\neg$ SN($\mathcal{R}$)
proof −

Clearly, minimal infinite chains are infinite chains.

have *chains*:
$\mathsf{ichain}_{\mathsf{min}}$ (DP($\mathcal{R}$), ID($\mathcal{R}$)) *s t* $\sigma \Longrightarrow$ ichain (DP($\mathcal{R}$), ID($\mathcal{R}$)) *s t* σ
by *simp*

By definition, we obtain that the signature of the initial DP problem is a subset of $\sharp(\mathcal{F}(\mathcal{R}))$.

have $\mathcal{F}$un(DP($\mathcal{R}$), ID($\mathcal{R}$)) $\subseteq \sharp(\mathcal{F}(\mathcal{R}))$ ⟨*proof omitted*⟩
from *ichain_imp_clean_ichain*[*OF this chains*[*OF assms*]]
have ichain (DP($\mathcal{R}$), ID($\mathcal{R}$)) *s t* $[\![\sigma]\!]_{\sharp(\mathcal{F}(\mathcal{R}))}$ **.**
from *clean_ichain_imp_not_SN_wfrstep*[*OF this*] **have** $\neg$ SN($\Rightarrow_{\mathcal{R}}$) **.**
thus *?thesis* **unfolding** *wfrstep_def SN_defs* **by** *blast*
qed

The main result for TRSs, is the proof of Theorem 7.1.

lemma *SN_wfrstep_SN_rstep_conv*: SN($\Rightarrow_{\mathcal{R}}$) = SN($\mathcal{R}$)

proof − **{**

As already said before, the direction from right to left, follows immediately from the definition of the well-formed rewriting relation.

assume ¬ SN($\Rightarrow\!\!\!\!\!\!\cdot\;_{\mathcal{R}}$)
hence ¬ SN($\mathcal{R}$) **unfolding** *SN_defs wfrstep_def* **by** *blast*
} moreover {

For the direction from left to right, we do a case distinction on whether $\mathcal{R}$ is well-formed or not.

assume ¬ SN($\mathcal{R}$)
have ¬ SN($\Rightarrow\!\!\!\!\!\!\cdot\;_{\mathcal{R}}$)
proof (*cases* WF($\mathcal{R}$))

If it is not well-formed, then clearly $\Rightarrow\!\!\!\!\!\!\cdot\;_{\mathcal{R}}$ is not terminating.

case *False* **thus** *?thesis* **by** *auto*
next

Otherwise, the proof follows the described outline.

case *True*
have $\mathcal{F}$un(DP($\mathcal{R}$), ID($\mathcal{R}$)) $\subseteq \sharp(\mathcal{F}(\mathcal{R}))$ ⟨*proof omitted*⟩
moreover from *not_SN_imp_min_ichain*[*OF True* ‹¬ SN($\mathcal{R}$)›]
obtain *s t σ* **where** ichain (DP($\mathcal{R}$), ID($\mathcal{R}$)) *s t σ* **by** *auto*
ultimately have ichain (DP($\mathcal{R}$), ID($\mathcal{R}$)) *s t* $[\![\sigma]\!]_{\sharp(\mathcal{F}(\mathcal{R}))}$
by (*rule ichain_imp_clean_ichain*)
from *clean_ichain_imp_not_SN_wfrstep*[*OF this*] **show** *?thesis* **.**
qed
}

After showing (the contrapositives of) both directions, we obtain the desired result.

ultimately show *?thesis* **by** *blast*
qed

7.3 Signature Extensions and Finiteness

Since the DP framework is concerned with DP problems instead of TRSs, we are also interested in how signature extensions affect the finiteness of DP problems. To get more fine-grained results, we do not restrict to finiteness alone, but consider also infinite chains that are not minimal.

We start by giving a negative result: cleaning may destroy minimal infinite chains.

Example 7.1. Consider the DP problem $(\mathcal{P}, \mathcal{R})$, where $\mathcal{P} = \{\mathsf{F}(\mathsf{g}(x, y)) \to \mathsf{F}(\mathsf{g}(x, y))\}$ and $\mathcal{R} = \{\mathsf{g}(x, x) \to \mathsf{g}(x, x)\}$. Then the sequences

$$\begin{aligned} s\ i &= \mathsf{F}(\mathsf{g}(x, y)) \\ t\ i &= \mathsf{F}(\mathsf{g}(x, y)) \\ \sigma\ i &= \{x/\mathsf{a}, y/\mathsf{b}\} \end{aligned}$$

constitute a minimal infinite chain, since $\mathsf{g}(\mathsf{a}, \mathsf{b})$ is terminating with respect to $\mathcal{R}$. However, when cleaning the substitution we get $[\![\sigma\ i]\!] = \{x/z, y/z\}$ and $\mathsf{g}(z, z)$ is not terminating with respect to $\mathcal{R}$. Hence, the chain is no longer minimal.

It has been shown by Sternagel and Thiemann [39], Example 13 that there are DP problems (originating from TRSs) admitting uncleaned minimal infinite chains, but no cleaned minimal infinite chain. Hence, signature extensions do in general not preserve finiteness of DP problems. In other words: restricting to just the function symbols occurring in a DP problem, allows to prove termination of a nonterminating TRS!

Note that there is an alternative definition of 'finiteness,' where only infinite chains (without the minimality condition) are considered. For this kind of finiteness, Lemma 7.1 shows that signature extensions preserve finiteness.

However, the definition of finiteness as given in Definition 5.5 is highly desirable in practice, since it admits powerful processors like the subterm criterion (Chapter 6), usable rules [12–14, 16, 46], and switching from full termination to innermost termination [12]; all of which are not sound without minimality. Thus, we are interested in conditions, under which signature extensions still

preserve finiteness. Left-linearity of the second component of a DP problem, is such a condition.

Definition 7.4 (Left-Linearity). A term is *linear*, if every single variable occurs at most once. That is,

$\mathsf{linear}(\mathit{Var}\ x) = \mathit{True}$
$\mathsf{linear}(\mathit{Fun}\ f\ ts) = (\mathit{is\_partition}\ [\mathcal{V}\mathsf{ar}(t).\ t \leftarrow ts] \wedge (\forall\, t{\in}ts.\ \mathsf{linear}(t)))$

where *is_partition* checks for a list of sets that they are mutually disjoint:

$\mathit{is\_partition}\ xs = (\forall\, j{<}|xs|.\ \forall\, i{<}j.\ xs_i \cap xs_j = \varnothing)$

A TRS is *left-linear*, if all its left-hand sides are linear.

$\mathsf{left\text{-}linear}(\mathcal{R}) \equiv \forall\,(l,\ r){\in}\mathcal{R}.\ \mathsf{linear}(l)$

Theorem 7.2. *Signature extensions preserve finiteness of a DP problem* $(\mathcal{P},\ \mathcal{R})$*, if* $\mathcal{R}$ *is left-linear. Or in Isabelle:*

$[\![\mathcal{F}\mathsf{un}(\mathcal{P},\ \mathcal{R}) \subseteq \mathcal{F};\ \mathsf{left\text{-}linear}(\mathcal{R});\ \mathsf{ichain}_{\mathsf{min}}\ (\mathcal{P},\ \mathcal{R})\ s\ t\ \sigma]\!]$
$\Longrightarrow \mathsf{ichain}_{\mathsf{min}}\ (\mathcal{P},\ \mathcal{R})\ s\ t\ [\![\sigma]\!]_{\mathcal{F}}$

Since the proof goes along the same lines as the proof of Lemma 7.1, we just present the auxiliary lemmas that ensure that minimality does not get lost, provided we have a left-linear $\mathcal{R}$.

First we need to show that rewrite steps propagate 'cleanness' for left-linear TRSs. (This is however only true, if we either have a well-formed TRS, or the starting term is terminating.)

Lemma 7.4. *Let* $\mathcal{R}$ *be a left-linear TRS,* $\mathcal{F}$ *a superset of the signature of* $\mathcal{R}$*,* s *and* t *terms, such that* s *is terminating with respect to* $\mathcal{R}$ *and there is an* $\mathcal{R}$*-step from* $[\![s]\!]_{\mathcal{F}}$ *to* t*. Then, there exists a term* u*, such that* $[\![u]\!]_{\mathcal{F}} = t$ *and* s *rewrites to* u*.*

lemma *left_linear_SN_elt_clean_term_rstep*:
assumes $[\![s]\!]_{\mathcal{F}} \rightarrow_{\mathcal{R}} t$ **and** $\mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}$ **and** $\mathsf{SN}_{\mathcal{R}}(s)$ **and** $\mathsf{left\text{-}linear}(\mathcal{R})$
shows $\exists\, u.\ [\![u]\!]_{\mathcal{F}} = t \wedge s \rightarrow_{\mathcal{R}} u$

We prove the lemma by induction over s. Here, *arbitrary*: t makes sure that the induction hypothesis is generalized to arbitrary terms instead of t.

using ‹$[\![s]\!]_{\mathcal{F}} \rightarrow_{\mathcal{R}} t$› **and** ‹$\mathsf{SN}_{\mathcal{R}}(s)$› **proof** (*induct s arbitrary*: *t*)

If s would be a variable, then $\mathcal{R}$ would contain a rule with a variable left-hand side. Hence s could not be terminating with respect to $\mathcal{R}$.

case (*Var x*) **thus** *?case* ⟨*proof omitted*⟩
next
case (*Fun f ss*)
note *IH* = *this*(*1*) **and** *rstep* = *this*(*2*)

From the induction hypothesis we obtain the two facts

IH: $[\![s \in ss; [\![s]\!]_{\mathcal{F}} \rightarrow_{\mathcal{R}} t; \mathsf{SN}_{\mathcal{R}}(s)]\!] \Longrightarrow \exists u.\ [\![u]\!]_{\mathcal{F}} = t \wedge s \rightarrow_{\mathcal{R}} u$
rstep: $[\![Fun\ f\ ss]\!]_{\mathcal{F}} \rightarrow_{\mathcal{R}} t$

show *?case*

We do a case distinction on whether cleaning will replace the current term by the fresh variable z, or not.

proof (*cases* $(f, |ss|) \in \mathcal{F}$)

If $[\![s]\!]_{\mathcal{F}}$ is z, then the proof proceeds as for the *Var*-case.

case *False* **show** *?thesis* ⟨*proof omitted*⟩
next

Otherwise, $[\![s]\!]_{\mathcal{F}}$ is of the form *Fun f* (*map* $[\![\cdot]\!]_{\mathcal{F}}$ *ss*).

case *True*
with *rstep* **have** (*Fun f* (*map* $[\![\cdot]\!]_{\mathcal{F}}$ *ss*)) $\rightarrow_{\mathcal{R}}$ *t* **by** *simp*
thus *?thesis*

We continue by a case distinction on whether the rewrite step takes place *at*, or *below* the root position.

proof (*cases rule*: *rstep_cases'*)
case (*root l r* σ)

Since $(l, r) \in \mathcal{R}$ and $\mathcal{R}$ is left-linear, we know that l is linear.

with ‹$(l, r) \in \mathcal{R}$› **and** ‹$\mathsf{left\text{-}linear}(\mathcal{R})$›

have $\mathsf{linear}(l)$ **by** (*auto simp*: *left_linear_trs_def*)

Moreover, as a left-hand side of a rule, l is well-formed with respect to $\mathcal{F}$.

moreover from ‹$(l, r) \in \mathcal{R}$› **and** ‹$\mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}$›
have $l \in \mathcal{T}(\mathcal{F})$ **using** *lhs_wf* **by** *blast*
moreover from *True* **and**‹$l\sigma = Fun\ f\ (map\ [\![\cdot]\!]_{\mathcal{F}}\ ss)$›
have $[\![Fun\ f\ ss]\!]_{\mathcal{F}} = l\sigma$ **by** *simp*

Using the lemma

linear_clean_subst:
$[\![\mathsf{linear}(t);\ t \in \mathcal{T}(\mathcal{F});\ [\![s]\!]_{\mathcal{F}} = t\sigma]\!] \Longrightarrow \exists \tau.\ [\![\tau|_{\mathcal{V}\mathsf{ar}(t)}]\!]_{\mathcal{F}} = \sigma|_{\mathcal{V}\mathsf{ar}(t)} \wedge s = t\tau$

we obtain a substitution τ whose cleaned version coincides with σ for all variables of l.

ultimately obtain τ **where** *clean_tau*: $[\![\tau|_{\mathcal{V}\mathsf{ar}(l)}]\!]_{\mathcal{F}} = \sigma|_{\mathcal{V}\mathsf{ar}(l)}$
and s: $Fun\ f\ ss = l\tau$
using *linear_clean_subst*[*OF* ‹$\mathsf{linear}(l)$›] **by** *blast*

The applied rule has to satisfy the variable condition, since otherwise $Fun\ f\ ss$ would not be terminating.

from ‹$\mathsf{SN}_{\mathcal{R}}(Fun\ f\ ss)$›
and *rhs_free_vars_imp_rstep_not_SN*[*OF* ‹$(l, r) \in \mathcal{R}$›]
have $\mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(l)$ **unfolding** s **by** *best*

Using this and several facts about substitutions (for example, the well-known coincidence lemma), we are able to derive $[\![r\tau]\!]_{\mathcal{F}} = t$. Thus, we may take $r\tau$ as our desired u.

have $[\![r\tau]\!]_{\mathcal{F}} = [\![r\tau|_{\mathcal{V}\mathsf{ar}(r)}]\!]_{\mathcal{F}}$ ⟨*proof omitted*⟩
also have ... $= [\![r\tau|_{\mathcal{V}\mathsf{ar}(l)}]\!]_{\mathcal{F}}$ ⟨*proof omitted*⟩
also have ... $= r[\![\tau|_{\mathcal{V}\mathsf{ar}(l)}]\!]_{\mathcal{F}}$ ⟨*proof omitted*⟩
also have ... $= r\sigma|_{\mathcal{V}\mathsf{ar}(l)}$ **unfolding** *clean_tau* **..**
also have ... $= r\sigma|_{\mathcal{V}\mathsf{ar}(r)}$ ⟨*proof omitted*⟩
also have ... $= r\sigma$ ⟨*proof omitted*⟩
finally have $[\![r\tau]\!]_{\mathcal{F}} = t$ **using** ‹$r\sigma = t$› **by** *simp*

moreover from ‹$(l, r) \in \mathcal{R}$› **have** $l\tau \to_{\mathcal{R}} r\tau$ **by** *best*
ultimately show *?thesis* **unfolding** s **by** *auto*
next

This leaves the case that the rewrite step takes place below the root.

case (*nonroot g ss1 u ss2 v*)

Let g be the root of t, u the argument of *Fun f* (*map* $[\![\cdot]\!]_F$ *ss*) that is reduced, *ss1* and *ss2* the surrounding arguments, and v the result of rewriting u. Clearly, the root of the term does not change. Further, u is the only argument that is changed.

hence $g = f$ **and** *args*: *map* $[\![\cdot]\!]_{\mathcal{F}}$ *ss* = *ss1* @ *u* # *ss2* **by** *auto*

For convenience, we introduce the following abbreviations. Let *?i* be the position of the reduced argument, *?ss1* the arguments in front, *?ss2* the arguments that follow, and *?C* the context of the rewrite step.

let *?i* = |*ss1*|
let *?ss1* = *take ?i ss*
let *?ss2* = *drop* (*Suc ?i*) *ss*
let *?C* = *More f ?ss1* □ *?ss2*
from *args* **have** |*map* $[\![\cdot]\!]_{\mathcal{F}}$ *ss*| = |*ss1* @ *u* # *ss2*| **by** *auto*
hence *len*: *?i* < |*ss*| **by** *auto*
from *args* **have** (*map* $[\![\cdot]\!]_{\mathcal{F}}$ *ss*)$_{?i}$ = (*ss1* @ *u* # *ss2*)$_{?i}$ **by** *auto*
with *len* **and** ‹$u \to_{\mathcal{R}} v$› **have** *step*: $[\![ss_{?i}]\!]_{\mathcal{F}} \to_{\mathcal{R}} v$ **by** *auto*
from *len* **have** $ss_{?i} \in ss$ **by** *auto*

If the whole term is terminating, then the *?i*-th argument is also terminating.

with *SN_imp_SN_arg*[*OF* ‹$\mathsf{SN}_{\mathcal{R}}$(*Fun f ss*)›]
  have *SN*: $\mathsf{SN}_{\mathcal{R}}(ss_{?i})$ **by** *auto*
from *IH*[*OF nth_mem*[*OF len*] *step SN*] **obtain** *w*
  where *v*: $[\![w]\!]_{\mathcal{F}} = v$ **and** *arg_step*: $ss_{?i} \to_{\mathcal{R}} w$ **by** *auto*
from *id_take_nth_drop*[*OF len*]
  have *len_ss*: |(*?ss1* @ $ss_{?i}$ # *?ss2*)| = |*ss*| **by** *simp*
have $[\![$*Fun f* (*?ss1* @ *w* # *?ss2*)$]\!]_{\mathcal{F}}$ =
  Fun f (*map* $[\![\cdot]\!]_{\mathcal{F}}$ *?ss1* @ *v* # *map* $[\![\cdot]\!]_{\mathcal{F}}$ *?ss2*)

```
      using True unfolding len_ss[symmetric] v[symmetric] by simp
    also have ... = t ⟨proof omitted⟩
    finally have ⟦Fun f (?ss1 @ w # ?ss2)⟧_F = t .
    moreover from arg_step have ?C[ss ?i] →_R ?C[w] ..
    ultimately show ?thesis
      unfolding ctxt_apply.simps
      unfolding id_take_nth_drop[OF len, symmetric] by best
  qed
 qed
qed
```

Now that we know that cleanness is propagated by rewriting (for left-linear TRSs), we can show that cleaning a term preserves termination (with respect to a left-linear TRS).

Lemma 7.5 (Cleaning Preserves Termination of Terms). *Let t be a term that is terminating with respect to $\mathcal{R}$. Whenever we clean t with respect to some superset of the signature of $\mathcal{R}$, the resulting term is still terminating with respect to $\mathcal{R}$.*

```
lemma left_linear_SN_elt_imp_clean_SN_elt:
 assumes left_linear: left-linear(R) and subset: F(R) ⊆ F and SN: SN_R(t)
 shows SN_R(⟦t⟧_F)
using ⟨SN_R(t)⟩ proof (rule contrapos_pp)
 assume ¬ SN_R(⟦t⟧_F)
 then obtain s where s 0 = ⟦t⟧_F
  and seq: ∀ i. (s i) →_R (s (i + 1)) by auto
```

We use the auxiliary function

$$S\ \mathcal{R}\ f\ t\ s\ 0 = t$$
$$S\ \mathcal{R}\ f\ t\ s\ (i+1) = (\mathit{SOME}\ u.\ S\ \mathcal{R}\ f\ t\ s\ i \to_{\mathcal{R}} u \wedge f\ u = s\ (i+1))$$

in order to construct a clean infinite sequence of $\mathcal{R}$-steps. Here, *SOME* x. $P\ x$ is very similar to $\exists\, x.\ P\ x$, only that instead of just *True* or *False*, *SOME*, will return a specific witness, if possible, and is undefined, otherwise.

```
 let ?s = S R ⟦·⟧_F t s
 let ?P = λi u. (?s i) →_R u ∧ ⟦u⟧_F = s (i + 1)
```

```
have ∀ i. (?s i) →_R (?s (i + 1)) ∧ [[?s (i + 1)]]_F = s (i + 1) ∧ SN_R(?s i)
proof
  fix i
  show (?s i) →_R (?s (i + 1)) ∧ [[?s (i + 1)]]_F = s (i + 1) ∧ SN_R(?s i)
  proof (induct i)
    case 0
    from seq have (s 0) →_R (s (0 + 1)) by simp
    hence [[t]]_F →_R (s (0 + 1)) by (simp add: ‹s 0 = [[t]]_F›)
    from left_linear_SN_elt_clean_term_rstep[OF this subset SN left_linear]
      have ∃ u. ?P 0 u by auto
    from someI_ex[OF this] have ?P 0 (?s (0 + 1)) by simp
    with SN show ?case by simp
  next
    case (Suc i)
    hence IH1: (?s i) →_R (?s (i + 1))
      and IH2: [[?s (i + 1)]]_F = s (i + 1)
      and IH3: SN_R(?s i) by auto
    from IH1 and IH3
      have SN': SN_R(?s (i + 1)) by (rule step_preserves_SN_elt)
    from seq have (s (i + 1)) →_R (s ((i + 1) + 1)) by simp
    hence [[?s (i + 1)]]_F →_R (s ((i + 1) + 1)) unfolding IH2 .
    from left_linear_SN_elt_clean_term_rstep[OF this subset SN' left_linear]
      have ∃ u. ?P (i + 1) u by auto
    from someI_ex[OF this] have ?P (i + 1) (?s ((i + 1) + 1)) by simp
    with SN' show ?case by simp
  qed
qed
moreover have ?s 0 = t by simp
ultimately show ¬ SN_R(t) by best
qed
```

Now, it is clear that the proof of Lemma 7.1 can be modified such that for left-linear $\mathcal{R}$, minimal infinite chains imply clean minimal infinite chains (since, cleaning preserves termination). Thus, we have a proof of Theorem 7.2.

7.4 Applications

The results about signature extensions are not only interesting from a theoretical point of view, but also important for some techniques that are used by many termination tools. We give two examples:

1. The first is a special case of term rewriting, *string rewriting*, where we restrict to TRSs that are built exclusively over unary function symbols (so called *string rewrite systems*; SRSs). There are techniques that work better/only for SRSs (to mention only two: arctic matrix interpretations [22] and string reversal). In the soundness proofs of those techniques, we rely on the fact that all function symbols are unary. We can do this, since it is safe to restrict to the function symbols that are occurring inside a system.
2. The second is a labeling transformation on TRSs or DP problems, *root-labeling*. It will be discussed in detail in Chapter 8.

7.5 Chapter Notes

We have given an alternative proof of the fact that signature extensions preserve the termination of TRSs. More importantly, this is the first mechanized proof of that fact. There are modularity results, due to Middeldorp [27], implying this fact. A shorter proof of the aforementioned modularity results was given by Ohlebusch [30]. In [39] we hint, why those proofs are not easily mechanizable. We further extended the results of the literature by investigating how signature extensions affect (minimal) infinite chains. Here, we have shown that signature extensions preserve finiteness of a DP problem, whenever the second component of the DP problem is left-linear.

Chapter 8

Root-Labeling

Failure is the key to success;
each mistake teaches us something.

Morihei Ueshiba

Besides termination techniques (that either reduce the number of rules in a problem or directly prove termination), there are also transformations which produce a new TRS whose termination implies termination of the original system (and which are hopefully easier to prove terminating). One of these transformations is *semantic labeling*, which uses semantic information to label different occurrences of function symbols in a TRS.

Example 8.1. Consider the following TRS, representing the factorial function (where p denotes taking the predecessor of a Peano number):

$$\begin{aligned} \mathsf{fact}(\mathsf{s}(x)) &\to \mathsf{mul}(\mathsf{fact}(\mathsf{p}(\mathsf{s}(x))), \mathsf{s}(x)) \\ \mathsf{p}(\mathsf{s}(x)) &\to x \end{aligned}$$

The intuitive meaning of the occurring function symbols is

$$\begin{aligned} \mathsf{fact}(x) &= x! \\ \mathsf{mul}(x, y) &= xy \\ \mathsf{p}(x) &= x - 1 \\ \mathsf{s}(x) &= x + 1 \end{aligned}$$

If we now label fact by the value of its argument, we get infinitely many rules of the form

$$\mathsf{fact}_{i+1}(\mathsf{s}(x)) \to \mathsf{mul}(\mathsf{fact}_i(\mathsf{p}(\mathsf{s}(x))), \mathsf{s}(x))$$

where i is the result of evaluating x (using the above interpretations). This version of the TRS, can easily be proved terminating by an LPO.

This should illustrate that interpreting the function symbols of a TRS in the "right way," helps to prove termination. However, it is not at all trivial to find the "right way." That is, why we are interested in purely syntactic interpretations. This leads us to root-labeling which is a specific instance of semantic labeling, where the used interpretation is fixed by the syntactic structure of the given system. In the following, we will first formally introduce semantic labeling in its model version (that is, the used interpretation has to reflect the actual rewrite rules, faithfully). Then, we will introduce root-labeling and show that the model condition is not always satisfied. Finally, we show a way to "preprocess" a TRS, such that the model condition is always satisfied.

8.1 Semantic Labeling

In this section, we recall the model version of semantic labeling. Usually, algebras are used to interpret TRSs.

Definition 8.1 (Algebras and Models). An *$\mathcal{F}$-algebra* $\mathcal{A}$ for $\mathcal{R}$ over some carrier set A, is given by the interpretation functions

$$f_{\mathcal{A}}(x_1, \ldots, x_n) : A^n \to A$$

for every function symbol $(f, n) \in \mathcal{F}$. A *(variable) assignment* is a mapping $\alpha : \mathcal{V} \to A$. This is extended to terms by

$$[\alpha]_{\mathcal{A}}(t) = \begin{cases} f_{\mathcal{A}}([\alpha]_{\mathcal{A}}(t_1), \ldots, [\alpha]_{\mathcal{A}}(t_n)) & \text{if } t = f(t_1, \ldots, t_n), \\ \alpha(t) & \text{otherwise} \end{cases}$$

We say that $\mathcal{A}$ is a *model* of $\mathcal{R}$, if $[\alpha]_{\mathcal{A}}(l) = [\alpha]_{\mathcal{A}}(r)$, for every rule $(l, r) \in \mathcal{R}$ and every assignment α.

Definition 8.2 (Labelings). A *labeling* ℓ for $\mathcal{A}$, consists of sets of labels L_f for every f, together with mappings $\ell_f : A^n \to L_f$, for every $(f, n) \in \mathcal{F}$ with $L_f \neq \varnothing$. Terms are labeled by

$$\mathsf{lab}_\alpha(x) = x$$
$$\mathsf{lab}_\alpha(f(t_1, \ldots, t_n)) = \begin{cases} f(\mathsf{lab}_\alpha(t_1), \ldots, \mathsf{lab}_\alpha(t_n)) & \text{if } L_f = \varnothing, \\ f_a(\mathsf{lab}_\alpha(t_1), \ldots, \mathsf{lab}_\alpha(t_n)) & \text{otherwise} \end{cases}$$

where $a = \ell_f([\alpha]_\mathcal{A}(t_1), \ldots, [\alpha]_\mathcal{A}(t_n))$. The *labeled TRS* $\mathcal{R}_{\mathsf{lab}}$, consists of the rewrite rule $\mathsf{lab}_\alpha(l) \to \mathsf{lab}_\alpha(r)$, for every rule $(l, r) \in \mathcal{R}$ and assignment $\alpha : \mathcal{V} \to A$.

The model version of semantic labeling is as follows.

Theorem 8.1 (Zantema [52], Theorem 4). *Let $\mathcal{R}$ be a TRS. Let the algebra $\mathcal{A}$ be a non-empty model of $\mathcal{R}$ and let ℓ be a labeling for $\mathcal{A}$. Then, $\mathcal{R}$ is terminating if and only if $\mathcal{R}_{\mathsf{lab}}$ is terminating.*

Here, the fact that $\mathcal{A}$ needs to be a (non-empty) model of $\mathcal{R}$, is called the *model condition*. Consider the following example to see that this condition is really necessary.

Example 8.2. Let $\mathcal{R}$ be a TRS consisting of the rules:

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

We interpret $\mathcal{R}$ over the Boolean algebra (where $1 + 1 = 0$): $\mathsf{f}_\mathcal{A}(x) = x$, $\mathsf{a} = 0$, and $\mathsf{b} = 1$. For the labeling, we take $L_\mathsf{f} = \{0, 1\}$ and $L_\mathsf{a} = L_\mathsf{b} = \varnothing$, together with the mapping $\ell_\mathsf{f}(x) = x + 1$. Then, the labeled system is

$$\mathsf{f}_1(\mathsf{a}) \to \mathsf{f}_0(\mathsf{b})$$
$$\mathsf{b} \to \mathsf{a}$$

having the single DP $\mathsf{F}_1(\mathsf{a}) \to \mathsf{B}$, and hence an empty dependency graph. Thus, by dropping the model condition, we could prove termination of a nonterminating TRS.

Semantic labeling was formalized for IsaFoR by René Thiemann. Furthermore, root-labeling is just an instance of semantic labeling. Hence, we do not give the formalization and its proofs within this thesis. Instead, we concentrate on the *closure under flat contexts*, which is the main obstacle for applying root-labeling effectively. But more on this, after an introduction to root-labeling.

8.2 Plain Root-Labeling

As mentioned in the introduction, we avoid the challenge of finding an appropriate carrier and suitable interpretation and labeling functions, by fixing everything beforehand, depending on the syntactic structure of the given TRS. Surely, this reduces the power of semantic labeling significantly, but we gain automation of the method for free.

Definition 8.3 (Root-Labeling of TRSs). Let $\mathcal{R}$ be a TRS over the signature $\mathcal{F}$. Then, the algebra $\mathcal{A}_\mathcal{F}$ has the carrier $\mathcal{F}$ and interprets function symbols by the mappings $f_{\mathcal{A}_\mathcal{F}}(x_1, \ldots, x_n) = f$ for all $(f, n) \in \mathcal{F}$. For the labeling we take $L_f = \mathcal{F}^n$ for every $(f, n) \in \mathcal{F}$, together with the mappings $\ell_f(x_1, \ldots, x_n) = (x_1, \ldots, x_n)$. (In words: every function symbol is interpreted by itself and we label by the tuple (internally we use lists) resulting from the interpretations of all arguments.) The labeled TRS is denoted by $\mathcal{R}_{\mathsf{rl}}$.

By Theorem 8.1, we know that whenever the interpretation functions of root-labeling constitute a non-empty model of a TRS $\mathcal{R}$ (which is not necessarily the case; cf. [35, Example 4]), then $\mathcal{R}_{\mathsf{rl}}$ is terminating if and only if $\mathcal{R}$ is terminating. We defer the problem of making sure, that the used interpretation constitutes a model to Section 8.4 and proceed by introducing root-labeling for DP problems.

8.3 Root-Labeling Processor

In order to benefit from the numerous termination techniques that are available in the DP framework, we found it worthwhile to reformulate root-labeling as a DP processor. Since, semantic labeling was already reformulated as a DP processor, this is an easy task. We mainly apply the labeling to $\mathcal{P}$ and $\mathcal{R}$ separately, in order to obtain the labeled DP problem $(\mathcal{P}_{\mathsf{rl}}, \mathcal{R}_{\mathsf{rl}})$. However, for the labeling of $\mathcal{P}$ we do not even need the model condition.

Definition 8.4 (Root-Labeling of DP Problems). Let $(\mathcal{P}, \mathcal{R})$ be a DP problem. By $\mathcal{F}_{\epsilon<}(\mathcal{P})$, we denote the function symbols in $\mathcal{P}$ that only occur below root positions. Then, the algebra $\mathcal{A}_\mathcal{G}$ has the carrier $\mathcal{G} = \mathcal{F}(\mathcal{R}) \cup \mathcal{F}_{\epsilon<}(\mathcal{P})$ and interprets function symbols as in Definition 8.3. For the labeling, we take $L_f = \mathcal{G}^n$ if $(f,\ n) \in \mathcal{G}$ with $n > 0$, and $L_f = \varnothing$, otherwise (this means that head symbols are not labeled). Now, the root-labeling processor just returns $(\mathcal{P}_{\mathsf{rl}}, \mathcal{R}_{\mathsf{rl}})$.

Recall that there is no model requirement for $\mathcal{P}$. However, root-labeling is only sound, when the used algebra forms a model of $\mathcal{R}$. The real challenge is to preprocess DP problems (and TRSs for that matter) in a way such that root-labeling will always constitute a model of $\mathcal{R}$. This is taken care of by the so called *closure under flat contexts*.

8.4 Closure Under Flat Contexts

Since for root-labeling, every function symbol is interpreted by itself, the model condition only depends on the root symbols of left-hand sides and right-hand sides. Now, by wrapping all the rules where the root of the left-hand side and the root of the right-hand side differ, inside an appropriate context, we can ensure the model condition for arbitrary TRSs and DP problems.

Definition 8.5 (Flat Contexts). A context that contains exactly one function symbol is called *flat*. For an n-ary function symbol f, the flat-context for the i-th argument is given by:

$$\mathcal{FC}^i_\mathcal{V}(f) \equiv \mathit{More}\ f\ (\mathit{take}\ i\ \mathcal{V})\ \Box\ (\mathit{drop}\ (i+1)\ \mathcal{V})$$

where $\mathcal{V}$ is an n-element list of 'fresh' variables. The set of *flat-contexts* built over the signature $\mathcal{F}$ is defined as follows:

$$\mathcal{FC}(\mathcal{F}) \equiv \bigcup_{(f,\ n)\in\mathcal{F}} \{\mathcal{FC}^i_{\mathsf{fresh}(n)}(f) \mid i \in \{0..{<}n\}\}$$

Here, by $\mathsf{fresh}(n)$, we denote a list of n 'fresh' variables and $\{0..{<}n\}$, is Isabelle's notation for a set, containing all natural numbers less than n. By freshness, we mean that those variables have not been used before (internally, we just make sure that the variables are distinct from those, occurring in the given DP problem).

For some TRSs, the model condition is satisfied automatically, since for every rule, the root of the left-hand side and the root of the right-hand side coincide. For others this is not the case and we have to consider the rules which have different root symbols (those are called *root-altering rules* and denoted by $\mathcal{R}_\mathsf{a}$) for preprocessing.

Definition 8.6 (Closure Under Flat Contexts of TRSs). The *closure under flat contexts* of a set of rules $\mathcal{R}$, with respect to a signature $\mathcal{F}$, is defined by:

$$\mathcal{FC}_\mathcal{F}(\mathcal{R}) \equiv (\bigcup_{(l,\ r)\in\mathcal{R}_\mathsf{a}} \{(C[l],\ C[r]) \mid C \in \mathcal{FC}(\mathcal{F})\}) \cup (\mathcal{R} - \mathcal{R}_\mathsf{a})$$

Example 8.3. Recall the TRS of Example 5.1. The signature consists of the symbols $\{(\mathsf{0}, 0), (\mathsf{s}, 1), (\mathsf{add}, 2)\}$, resulting in the following set of flat contexts:

$$\{\mathsf{s}(\Box), \mathsf{add}(\Box, x_2), \mathsf{add}(x_1, \Box)\}$$

Both rules are root-altering and hence need to be closed under flat contexts. The resulting closed TRS consist of the six rules:

$$\begin{aligned}
\mathsf{s}(\mathsf{add}(\mathsf{0}, y)) &\to \mathsf{s}(y)\\
\mathsf{s}(\mathsf{add}(\mathsf{s}(x), y)) &\to \mathsf{s}(\mathsf{s}(\mathsf{add}(x, y)))\\
\mathsf{add}(\mathsf{add}(\mathsf{0}, y), x_2) &\to \mathsf{add}(y, x_2)\\
\mathsf{add}(\mathsf{add}(\mathsf{s}(x), y), x_2) &\to \mathsf{add}(\mathsf{s}(\mathsf{add}(x, y)), x_2)\\
\mathsf{add}(x_1, \mathsf{add}(\mathsf{0}, y)) &\to \mathsf{add}(x_1, y)\\
\mathsf{add}(x_1, \mathsf{add}(\mathsf{s}(x), y)) &\to \mathsf{add}(x_1, \mathsf{s}(\mathsf{add}(x, y)))
\end{aligned}$$

Clearly, after closing a TRS under flat contexts, the model condition is always satisfied. However, we need to make sure that the termination behavior of the system is not disturbed.

For TRS, we have the following result:

Theorem 8.2. *The closure under flat contexts for TRSs is termination preserving and reflecting. Or in Isabelle:*

$$[\![\mathit{finite}\ \mathcal{R};\ \mathcal{F}(\mathcal{R}) \subseteq \mathcal{F};\ \mathcal{FC}(\mathcal{F}) \neq \varnothing]\!] \Longrightarrow \mathsf{SN}(\mathcal{R}) = \mathsf{SN}(\mathcal{FC}_\mathcal{F}(\mathcal{R}))$$

Note the side conditions: this only works for finite TRSs (meaning that we only consider a finite number of rules), when considering at least all the function

symbols of the signature of $\mathcal{R}$, and when there is at least one flat context. The finiteness of $\mathcal{R}$ is required in order to make sure that at most finitely many variable names have already been used inside the TRS. This is a technical precondition for internally used *fresh name generators* that allow a list (and hence only a finite number) of names that have to be avoided. The condition that the set of flat contexts is non-empty, implies that there has to be at least one non-constant function symbol in the signature of $\mathcal{R}$. Since systems, consisting only of constants cannot be labeled, this is not a real restriction.

Hence, for TRSs, the closure under flat contexts followed by root-labeling, is a sound and complete termination technique.

Next, we consider DP problems instead of TRSs. In [35, Lemmas 13 and 17], we claimed that the closure and flat contexts is sound for DP problems. However, in both proofs (which where not formalized then), we have a "*without loss of generality*," where we unfortunately lose generality. This came to our attention, when we formalized the closure under flat contexts in Isabelle. The problem is that the operation depends on signature extensions (since otherwise, we could not restrict the set of flat contexts to just those symbols, occurring in the DP problem). As has been shown in Chapter 7, signature extensions are only sound for left-linear $\mathcal{R}$.

Before we rectify the wrongly stated theorem, we recall the definition of closing DP problems under flat contexts.

Definition 8.7 (Closure Under Flat Contexts of DP Problems). For DP problems ($\mathcal{P}$, $\mathcal{R}$), we do not just close $\mathcal{P}$ and $\mathcal{R}$ under flat contexts, since this would result in having function symbols of $\mathcal{R}$ at root positions in the closed $\mathcal{P}$. In order to preserve termination we need a trick (the problem are $\mathcal{R}$-steps that take place directly below sharp symbols). Consider the auxiliary function

$$\mathsf{block}_\Delta(\mathit{Var}\ x) = \mathit{Var}\ x$$
$$\mathsf{block}_\Delta(\mathit{Fun}\ f\ ts) = \mathit{Fun}\ f\ (\mathit{map}\ (\lambda t.\ \mathit{Fun}\ \Delta\ [t])\ ts)$$

which does insert the function symbol Δ, directly below the root at every argument position. Now, the closure under flat contexts of ($\mathcal{P}$, $\mathcal{R}$), is defined by

$$(\mathsf{block}_\Delta(\mathcal{P}),\ \mathcal{FC}_{\{\Delta\}\ \cup\ \mathcal{F}}(\mathcal{R}))$$

where $\mathsf{block}_\Delta(\mathcal{P}) = \{(\mathsf{block}_\Delta(s),\ \mathsf{block}_\Delta(t)) \mid (s,\ t) \in \mathcal{P}\}$, $\mathcal{F}$ is a superset of

$\mathcal{F}_{\epsilon<}(\mathcal{P}) \cup \mathcal{F}(\mathcal{R})$, and Δ is assumed to be fresh with respect to the signature of $\mathcal{R}$.

Thus, for DP problems, we have the following result:

Theorem 8.3. *Let $(\mathcal{P}, \mathcal{R})$ be a DP problem consisting of finitely many rules and $\mathcal{F}_{\epsilon<}(\mathcal{P}) \cup \mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}$. Whenever $\mathcal{R}$ is left-linear, then the processor returning $(\mathsf{block}_\Delta(\mathcal{P}), \mathcal{FC}_{\{\Delta\} \cup \mathcal{F}}(\mathcal{R}))$ is sound.*

We refrain from presenting the proof, since it is very technical. The main idea is the same as in the paper-proof in [35]. The interested reader is referred to the theory *RootLabeling* of IsaFoR.

8.5 Touzet's SRS

The main example of [35] was an SRS, introduced by Touzet [44]:

$\mathsf{bu} \to \mathsf{bs}$	$\mathsf{sbs} \to \mathsf{bt}$	$\mathsf{tb} \to \mathsf{bs}$	$\mathsf{ts} \to \mathsf{tt}$
$\mathsf{sb} \to \mathsf{bsss}$	$\mathsf{su} \to \mathsf{ss}$	$\mathsf{tbs} \to \mathsf{utb}$	$\mathsf{tu} \to \mathsf{ut}$

In version 7.0.2 of the termination problem database (TPDB)[1] this system has the name `TRS/Zantema_04/z090.xml`. By using root-labeling in the DP framework (and preceded by the closure under flat contexts, since labeling would not be sound otherwise) we could generate the first (and still only) automatic termination proof of `TRS/Zantema_04/z090.xml`. Gladly, the system is left-linear (since it is an SRS) and hence our formalization even yields an automatic certificate of its termination. The rather longish (over 5 MB) proof is to be found at

`http://cl-informatik.uibk.ac.at/software/ceta/z090.proof.xml`

and can be automatically certified by version 1.15 of CeTA.

[1] `http://termcomp.uibk.ac.at/status/downloads/tpdb-7.0.2.tar.gz`

8.6 Chapter Notes

Semantic labeling was introduced by Zantema [52]. A very special version of semantic labeling for string rewrite systems, due to Johannes Waldmann, in which the semantic and labeling components are completely determined by the system at hand, was first used by Waldmann [47] and Endrullis [10], in the string rewriting division of the 2006 international termination competition[2] with remarkable success. This special version—which was later called *root-labeling*—was extended to TRSs and also formulated as a DP processor by Sternagel and Middeldorp [35]. Actually, [35] introduces two different DP processors for root-labeling. Our formalization corresponds to $\mathcal{FC}_1$ of [35]. During our formalization, we found an error in [35, Lemmas 13 and 17], which was rectified and formalized in Isabelle by Sternagel and Thiemann [39].

[2] `www.termination-portal.org/wiki/Termination_Competition`

Chapter 9

Certification

Security, like correctness, is not an add-on feature.

Andrew Stuart Tanenbaum

Besides black art, there is only automation and mechanization.

Federico García Lorca

Until now we have only dealt with the first stage of building a termination proof checker as described in Chapter 1. Of this first stage, we have concentrated on formalizing the mathematical theory that is used in termination proofs. Our goal, however, is to certify for a given termination proof that it is correct. Hence, for every technique that is used in such a proof, we need a way to check for a correct application of this technique and additionally, we need to check that all incorporated techniques are composed correctly. In the following, we describe how we handle this task in IsaFoR/CeTA.

9.1 Proof Format

The first ingredient, is a general proof format, such that every termination tool can generate termination proofs using this format and every termination proof checker can certify (or reject) proofs in this format. For the certification of termination proofs, the ***C****ertification* ***P****roof* ***F****ormat* has been developed.[1] This is an XML format that contains a description of a problem (TRS,

[1] `http://cl-informatik.uibk.ac.at/software/cpf/`

DP problem, etc.) together with a proof of some desired property (termination/nontermination, finiteness, etc.). The details are not important here. Just note that there is such a format and that we have implemented parser combinators in Isabelle that are able to transform this XML format into an Isabelle data type for termination proofs.

We represent termination proofs of TRSs by the type (α, β, γ) *TRSProof*. Here, α is the type of function symbols, β the type of labels, and γ the type of variables. Further, for every termination technique that is supported by CeTA, there is a constructor. We only present those constructors that represent one of the techniques that where described in previous chapters. For TRSs, this includes *DPTrans* for the DP transformation, *Fcc* for the closure under flat contexts, and *Rootlab* for root-labeling.

For proofs inside the DP framework, we use the type (α, β, γ) *DPProof*. We again have a constructor for every supported processor. For example, *SubtermCriterionProc* for the subterm criterion processor, *FccProc* for the closure under flat contexts of DP problems, and *RootlabProc* for the root-labeling processor.

9.2 Check Functions

The actual certification of a given termination proof is done as follows:

1. Translate the input into the internal data type (α, β, γ) *TRSProof*.
2. Call a check function that either succeeds, or otherwise fails, providing an informative error message.

For the second step, every termination technique (for TRSs) and every processor (for DP problems), has its own check function. Then, the 'global' check function for TRS proofs, merely calls the specific check functions according to the structure of the given proof. In order to obtain readable error messages on failure, our check functions have the result type ε *check*, which is an abbreviation for ε + *unit*, that is, every check function either returns *unit* (which indicates success), or some value of type ε (which is then used to produce an error message). In the following, we describe the check functions of some termination techniques.

9.2.1 DP Transformation

Checking the correct application of the DP transformation, is done using the function

$$
\begin{array}{l}
\mathit{check\_DP}\ \mathcal{R}\ \mathcal{P} \equiv \mathsf{do}\ \{ \\
\quad \mathit{check\_wf\_trs}\ \mathcal{R}; \\
\quad \mathit{check\_subseteq}\ (\mathit{DP\_list}\ \mathcal{R}) \\
\quad\ \mathcal{P} \leftsquigarrow (\lambda x.\ \text{“the DP ”} \cdot \\
\qquad\qquad\qquad \mathit{shows\_rule}\ x \cdot \text{“ does not appear in the DP problem”} \cdot \mathit{shows\_nl}) \\
\} \leftsquigarrow (\lambda x.\ \text{“the DP}-\text{transformation is not applied correctly.”} \cdot \mathit{shows\_nl} \cdot x)
\end{array}
$$

where *check_wf_trs* is a check function, guaranteeing that the given TRS is well-formed and *check_subseteq* checks whether all elements of its left argument are also contained in its right argument (here, this guarantees that none of the dependency pairs has been omitted; still, we allow for additional rules, as could for example result from a weaker definition of dependency pairs). The function *DP_list*, computes the dependency pairs for a given list of rewrite rules (on this level, we are using lists instead of sets, since our goal is to produce executable functions). Note, that this function uses the so called *do-notation* as syntactic sugar for monadic functions. The remaining constructs are for error messages: $\cdot$ just concatenates messages, whereas $\leftsquigarrow$ modifies an error message (its right argument) that is issued whenever its left argument fails.

Reasoning about the Error Monad.
Concerning $\cdot$, we are cheating a bit, since there are actually two different functions that we both represented by $\cdot$ here. For efficiency reasons (that is, to avoid a quadratic time complexity for the concatenation of error messages), we use the type *shows* (which is an abbreviation for *string* $\Rightarrow$ *string*) for error messages. This is similar to Haskell's `Show` type class. Then, there are two functions to modify messages. The first is *op* +#+ :: *string* $\Rightarrow$ *shows* $\Rightarrow$ *shows*, which is used to add a given *string* to an error message. The second is *op* +@+ :: *shows* $\Rightarrow$ *shows* $\Rightarrow$ *shows*, and is use to concatenate two error messages. Both are represented by $\cdot$ in the text.

The ASCII version of $\leftsquigarrow$ is *op* <+?.

The soundness of the DP transformation is expressed by the lemma

$[\![isOK\ (check\_DP\ \mathcal{R}\ \mathcal{P});\ \mathsf{finite}(\mathcal{P},\ \mathsf{ID}(\mathcal{R}))]\!] \Longrightarrow \mathsf{SN}(\mathcal{R})$

which constitutes the connection between the abstract formalization of the DP transformation (where we use abstract mathematical concepts like sets, quantifiers, etc.) and the concrete check function for given problems (using lists and only functions that can be expressed as functional programs). For completeness, we give the definition of the function *isOK*:

$isOK\ m \equiv \mathsf{case}\ m\ \mathsf{of}\ Inl\ e \Rightarrow False \mid Inr\ x \Rightarrow True$

9.2.2 Subterm Criterion

A correct application of the subterm criterion is checked via

$check\_subterm\_criterion\_proc\ \pi\ rsteps\ \mathcal{P}\ \mathcal{P}'\ \mathcal{R}\ (Some\ rm) = \mathsf{do}\ \{$

$forallm$

$(\lambda(l,\ r).\ \mathsf{do}\ \{$

$check\_no\_var\ l;$

$check\_no\_var\ r;$

$check\_no\_defined\_root\ rm\ r$

$\})$

$\mathcal{P} \leftsquigarrow snd;$

$forallm\ (check\_strict\ \mathcal{R}\ rsteps\ \pi)\ (list\_diff\ \mathcal{P}\ \mathcal{P}') \leftsquigarrow snd;$

$forallm\ (check\_weak\ \pi)\ \mathcal{P}' \leftsquigarrow snd$

$\}$

Here, π is the projection function, *rsteps* is a representation of the concrete rewrite sequences that are used for those rules which are in $(\rhd \cup \to_{\mathcal{R}})^+{}_{\pi}$, $\mathcal{P}$ are the pairs of the given DP problem, $\mathcal{P}'$ are the remaining pairs (after the application of the processor), and $\mathcal{R}$ are the rules of the given DP problem. Further, *rm* is a *rule map* (of the rules in $\mathcal{R}$) associating function symbols to rules, defining them. This is used for efficiently checking whether some symbol is defined with respect to $\mathcal{R}$. The *forallm* function, applies a check function to

all elements of a list. On error, it does not only return an error message, but also the element (together with its position in the list) which caused the error. Since here, we are only interested in the message, we apply *snd* to the resulting error. The check function proceeds as follows: First we compute the list of all defined symbols of $\mathcal{R}$. Then, we check for every pair in $\mathcal{P}$ that the left-hand side, as well as the right-hand side is not a variable, and additionally that the root of the right-hand side is not one of the defined symbols of $\mathcal{R}$ (ensuring the first assumption of Theorem 6.4). Afterwards, we check that all the removed rules (that is $\mathcal{P} - \mathcal{P}'$) are in $(\triangleright \cup \to_{\mathcal{R}})^+{}_\pi$ (ensuring the second assumption of Theorem 6.4). Finally, we check that all the remaining rules are equal after applying the projection (ensuring the third assumption of Theorem 6.4).

The link to the abstract formalization is given by the lemma:

$$
\begin{aligned}
&\mathit{isOK}\\
&\quad(\mathit{check\_subterm\_criterion\_proc}\ \pi\ \mathit{rsteps}\ \mathcal{P}\ \mathcal{P}'\ \mathcal{R}\\
&\qquad(\mathit{computeRuleMap}\ \mathcal{R}))\Longrightarrow\\
&\mathsf{chain\text{-}id}\ (\mathcal{P},\mathcal{R})\ (\mathcal{P}',\mathcal{R})
\end{aligned}
$$

That is, the remaining DP problem is chain-identifying with respect to the original DP problem.

9.2.3 Closure Under Flat Contexts

Checking the correct application of the closure under flat contexts in the DP framework, is a bit more involved. The termination tool has to provide a fresh function symbol f and the list of all flat contexts *fcs*, in addition to the incoming DP problem ($\mathcal{P}$, $\mathcal{R}$) well as the outgoing DP problem ($\mathcal{P}'$, $\mathcal{R}'$).

$$
\begin{aligned}
&\mathit{fcc\_proc}\ \mathcal{P}\ \mathcal{R}\ f\ \mathit{fcs}\ \mathcal{P}'\ \mathcal{R}' = \mathsf{do}\ \{\\
&\quad \mathsf{let}\ \mathit{fa} = (f,\ 0+1);\\
&\quad \mathsf{let}\ \mathit{Cf} = \mathit{More}\ f\ []\ \Box\ [];\\
&\quad \mathsf{let}\ \mathit{fcs}' = \mathit{Cf}\ \#\ \mathit{fcs};\\
&\quad \mathsf{let}\ \mathit{vs} = \mathit{vars\_trs\_list}\ \mathcal{R};
\end{aligned}
$$

```
  let fs = list_union (funas_below_trs_list P) (funas_trs_list R);
  let fas = fa # fs;
  let ds = defs_list (map_trsI remove_lab R);
  check (¬ member ds (remove_lab f, 1))
   (shows (remove_lab f) · " is not fresh" · shows_nl);
  check_left_linear_trs R;
  check_wf_trs R;
  forallm
   (λr. do {
         check_no_var (fst r);
         check_no_var (snd r);
         check_no_defined_root_defs ds (remove_lab_term (snd r))
       })
   P ↜ snd;
  forallm (check_flat_ctxt vs) fcs' ↜ snd;
  forallm (check_is_flat_ctxt vs fas) fcs' ↜ snd;
  forallm (check_flat_ctxt_complete fcs') fas ↜ snd;
  forallm (check_rule_preserving fcs' R') R ↜ snd;
  forallm (check_rule_reflecting fcs' R) R' ↜ snd;
  check_superset_of_blocked f P' P
}
```

In an initial phase, the check function computes several auxiliary values: the defined function symbols of $\mathcal{R}$, a unary function symbol *fa* (corresponding to Δ from the abstract formalization), a 'unary' flat context *Cf*, an extended list of flat context *fcs'*, the variables of $\mathcal{R}$, all the function symbols that do not occur at the roots of left-hand sides or right-hand sides in $\mathcal{P}$, and finally,

the list of function symbols that have to be considered when constructing flat contexts *fas*. Subsequently, we check that *fa* is fresh with respect to $\mathcal{F}_{\epsilon<}(\mathcal{P}) \cup \mathcal{F}(\mathcal{R})$ and that it is not defined in $\mathcal{R}$. Then, we check that $\mathcal{R}$ is left-linear and well-formed. For $\mathcal{P}$ all the left-hand sides and right-hand sides have to be non-variable terms and the right-hand sides must not be defined in $\mathcal{R}$. The next three checks, essentially make sure that the set *fcs'* really contains flat contexts and that it contains all of them. Finally, we check that $\mathcal{R}'$ and $\mathcal{P}'$ do not lack any rules.

Again, we have formalized the soundness of the flat context closure processor:

$[\![isOK\ (fcc\_proc\ \mathcal{P}\ \mathcal{R}\ \Delta\ fcs\ \mathcal{P}'\ \mathcal{R}'); \mathsf{finite}(\mathcal{P}', \mathcal{R}')]\!] \Longrightarrow \mathsf{finite}(\mathcal{P}, \mathcal{R})$

Note that the closure under flat contexts is not chain-identifying (and hence unlabeling is forbidden afterwards). The problem is that the infinite chain is modified by cleaning and by inserting function symbols directly below the root. Thus, we loose the tight connection that is needed for chain-identifyingness.

9.3 Code-Extraction

In a similar way, the individual check functions of the separate termination techniques are all linked to our abstract formalizations. In contrast to the abstract mathematical results, however, the functions are not at all different from usual functions in functional programming languages (that is, everything is executable). In contrast to most functions that are used in programs, we have formally proven that our check functions are correct. Further, there is a "main" function that manages an initially given proof and recursively calls the corresponding check functions to verify the application of every occurring termination technique or DP processor. Now we can use Isabelle's code-generation facilities, in order to extract code for a specific programming language (for example, Haskell, StandardML, or OCaml). The result, is a fully verified program (CeTA) that can efficiently check a given termination proof.

9.4 Chapter Notes

In this chapter, we have shown, how we use our abstract formalizations on the termination of rewriting, in order to certify given termination proofs. This is achieved by proving properties of check functions, which are suitable for code-generation.

A detailed account of code-generation in Isabelle is given by Haftmann [15].

Chapter 10

Conclusion

You know a conjurer gets no credit when once he has explained his trick; and if I show you too much of my method of working, you will come to the conclusion that I am a very ordinary individual after all.

Sherlock Holmes
A Study in Scarlet

In the last few chapters, we have given an overview of our most prominent contributions to IsaFoR/CeTA. The basic idea was to formalize facts about termination, as well as check functions that certify the correct application of these facts in specific proofs. This was done using Isabelle/HOL (IsaFoR). Then, we used code-extraction to obtain an efficient proof checker for termination proofs of term rewrite systems (CeTA).

We started by giving a brief introduction to interactive theorem proving (using Isabelle/HOL) in Chapter 2. Then, we presented our underlying formalizations of abstract rewriting as well as term rewriting in Chapters 3 and 4. The remainder of the thesis was mostly concerned with specific termination techniques. Switching from termination of term rewrite systems to finiteness of dependency pair problems was the topic of Chapter 5. This was followed by a formalization of the (generalized) subterm criterion—a simple, yet powerful processor for the dependency pair framework—in Chapter 6. As a prerequisite for later formalizations, we then made a short digression on the effect of signature extensions for termination of rewriting and finiteness of dependency pair problems, in Chapter 7. Afterwards, in Chapter 8, we discussed a crucial ingredient of root-labeling: the closure under flat contexts. Finally, in Chapter 9, we gave an overview on how our abstract formalizations are linked to check

functions for concrete applications of termination techniques in IsaFoR. Those check functions were then used to obtain the efficient proof checker CeTA, a program to certify the correctness of a given termination proof.

As already mentioned in the introduction, I just picked those parts of IsaFoR for my presentation that were mainly realized by myself. For completeness, we now give a list of techniques that can be certified by the current version of CeTA (which is 1.15):

- For termination those are: dependency pair transformation; dependency graph processor (EDG***); reduction pair processors in several variants; polynomial interpretations over naturals, rationals, arctic numbers, and over matrices of all these domains; lexicographic path order; usable rules; subterm criterion; semantic labeling; closure under flat contexts; root-labeling; size-change termination; and string reversal. (Recently, also support for relative termination has been added.)
- For nontermination those are: loops; non-well-formed term rewrite systems; string reversal; rule removal; and dependency pair transformation.

Before we present related work, we list IsaFoR/CeTA relevant publications: Sternagel and Thiemann [36, 37, 38, 39], Sternagel et al. [40], Thiemann and Sternagel [43], and Zankl et al. [51].

10.1 Related Work

We are aware of two other projects that concentrate on certifying termination proofs of term rewrite systems:

- Coccinelle/C*i*ME: As part of the A3PAT project,[1] Coccinelle is a Coq[2] library on rewriting and C*i*ME (Alt-Ergo) a tool, which (among other things) generates Coq proofs from given termination proofs, using the formalizations of Coccinelle. The A3PAT team came up with the generalized subterm criterion. More details on Coccinelle/C*i*ME may be found in Contejean et al. [6, 7] and Courtieu et al. [8].

[1] `http://a3pat.ensiie.fr`
[2] `http://coq.inria.fr`

- CoLoR/Rainbow: Here, CoLoR[3] is a Coq library on rewriting and Rainbow a program that generates Coq proofs from given termination proofs, using the formalizations of CoLoR. A distinctive feature of CoLoR is a formalization of the first- and higher-order recursive path ordering. Many details related to CoLoR/Rainbow have been given by Blanqui and Koprowski [4], Blanqui et al. [5], Koprowski [20, 21], and Koprowski and Zantema [23].

In both of the above approaches, a given termination proof is handled in two stages: first the proof is used as a recipe for generating a Coq script (using the respective libraries Coccinelle or CoLoR) that certifies termination of the given problem. Then, Coq is run on those scripts to check their correctness. The main difference to IsaFoR/CeTA, is that CeTA directly checks the correctness of a given proof (instead of generating a theory, to be run through Isabelle). This allows for graceful error handling. That is, CeTA gives informative error messages (using notions from rewriting), whereas in the other two approaches, Coq just fails with some internal error, if the generated Coq script cannot be handled. Furthermore, CeTA is code-generated and hence runs as binary program. Running a generated Coq script, is comparatively slow.

10.2 Applications and Future Work

Aside from the obvious application of certifying termination proofs of term rewrite systems, we designed the basic layer of IsaFoR in a way that it should be appropriate for other formalizations about rewriting. For example, we are planning to tackle termination under specific strategies. Furthermore, there is a new project on certification of confluence of term rewrite systems that will be based on IsaFoR. Another direction of research (that has already started), is to tighten the connection between Isabelle/HOL and CeTA, such that Isabelle/HOL functions may be proven terminating using CeTA.

[3] `http://color.inria.fr`

	# proofs		# supported		# certificates	
SRS Relative	33	(31 %)	18	(55 %)	18	(55 %)
SRS Standard	220	(76 %)	184	(84 %)	184	(84 %)
TRS Relative	15	(75 %)	14	(93 %)	11	(73 %)
TRS Standard	375	(89 %)	282	(75 %)	281	(75 %)

Table 10.1: Termination Competition 2010

10.3 Assessment

In order to get an impression of the current state-of-the-art in the automatic certification of termination proofs, we shortly present some results from the most recent international termination competition (which was conducted in July 2010).[4] There have been four categories for which we can in principle certify proofs: SRS Relative (containing 105 problems), SRS Standard (containing 289 problems), TRS Relative (containing 20 problems), and TRS Standard (containing 423 problems). In the certifying counterpart of all four of those categories, CeTA made the first place. Note that in the two relative categories, CeTA has been the only participant and unfortunately, not even all of the existing certifiers took part in the competition. However, the outcome would most certainly have been the same. In Table 10.1 we give a more detailed overview of CeTA's results. In the column '# proofs' we give the number of successful proofs (that is, some termination tool answering either YES or NO). Here, the percentage is computed with respect to the total number of problems in the corresponding category. The column '# supported' gives the number of successful proofs that where produced by termination tools, when restricted to just those techniques that are supported by at least one of the certifiers. Here and in the last column, the percentage is computed with respect to the first column. Finally, '# certificates' gives the number of proofs that have been accepted by CeTA.

It can be seen that for the standard categories, the gap between 'automatically provable' and 'automatically provable and certifiable', is less then 25 %. Note that without CeTA, that is, only considering the results of the second participant cime3verifier, the percentages would have been: 21 % for SRS Standard and

[4] `http://termcomp.uibk.ac.at`

42 % for TRS Standard certificates (corresponding to the last column of the above table).

Appendix A

Auxiliary Proofs

Most of the proofs that we have given in previous chapters rely on some auxiliary results. In cases, where those results are too lengthy or do not add to the understanding of a proof, we moved those auxiliary lemmas into the appendix. The following proofs are subdivided corresponding to the chapters for which they provide results.

A.1 General

If the range of an infinite sequence is finite, then there exists an element that occurs infinitely often.

lemma *finite_range*:
  fixes f::$nat \Rightarrow \alpha$
  assumes *finite* (*range* f) **shows** $\exists\, x.\ \exists_\infty i.\ f\ i = x$
proof (*rule ccontr*)
  assume $\neg(\exists\, x.\ \exists_\infty i.\ f\ i = x)$
  hence $\forall\, x.\ \exists\, j.\ \forall\, i{>}j.\ f\ i \neq x$ **unfolding** *INFM_nat* **by** *blast*
  from *choice*[*OF this*] **obtain** j **where** *neq*: $\forall\, x.\ \forall\, i{>}j\ x.\ f\ i \neq x$ **..**
  from *finite_range_imageI*[*OF assms*]
    have *finite* (*range* $(j \circ f)$) **by** (*simp add*: *comp_def*)
  from *finite_nat_bounded*[*OF this*] **obtain** m
    where *range* $(j \circ f) \subseteq \{..{<}m\}$ **..**
  hence $j\ (f\ m) < m$ **by** (*auto simp*: *comp_def*)
  with *neq* **have** $f\ m \neq f\ m$ **by** *auto*
  thus *False* **by** *simp*
qed

A.2 Abstract Rewriting

If we have an infinite sequence over the union of two ARSs, such that the first element of our sequence is terminating with respect to the second ARS, then, after a (possibly empty) finite sequence of steps in the second ARS, we will finally reach a step in the first ARS.

lemma *union_iseq_SN_elt_imp_first_step*:
  assumes $\forall\, i.\ (S\ i,\ S\ (i+1)) \in (\mathcal{A} \cup \mathcal{B})$ **and** $\mathsf{SN}_{\mathcal{B}}(S\ 0)$
  shows $\exists\, i.\ (S\ i,\ S\ (i+1)) \in \mathcal{A}$
    $\wedge\ (\forall\, j{<}i.\ (S\ j,\ S\ (j+1)) \in \mathcal{B} \wedge (S\ j,\ S\ (j+1)) \notin \mathcal{A})$
proof −
  from ‹$\mathsf{SN}_{\mathcal{B}}(S\ 0)$› **obtain** i **where** $(S\ i,\ S\ (i+1)) \notin \mathcal{B}$ **by** *blast*
  with *assms* **have** $(S\ i,\ S\ (i+1)) \in \mathcal{A}$ (**is** *?P i*) **by** *auto*
  let *?i* = *Least ?P*
  from ‹*?P i*› **have** *?P ?i* **by** (*rule LeastI*)
  have $\forall\, j{<}$ *?i*. $(S\ j,\ S\ (j+1)) \notin \mathcal{A}$ **using** *not_less_Least* **by** *auto*
  moreover with *assms* **have** $\forall\, j{<}$ *?i*. $(S\ j,\ S\ (j+1)) \in \mathcal{B}$ **by** *best*
  ultimately have $\forall\, j{<}$ *?i*. $(S\ j,\ S\ (j+1)) \in \mathcal{B} - \mathcal{A}$ **by** *best*
  with ‹*?P ?i*› **show** *?thesis* **by** *best*
qed

If we have an infinite sequence in some ARS, then every element of the sequence is connected to every later element of the sequence, by finitely many steps in the ARS.

lemma *steps_imp_seq*:
  fixes *i j*::*nat*
  assumes $\forall\, i.\ (S\ i,\ S\ (i+1)) \in \mathcal{A}$ **and** $j \geq i$ **shows** $(S\ i,\ S\ j) \in \mathcal{A}^*$
using ‹$j \geq i$› **proof** (*induct j*)
  case *0* **thus** *?case* **by** *simp*
next
  case (*Suc j*)
  show *?case*
  proof (*cases* $i = (j+1)$)
    case *True* **show** *?thesis* **by** (*simp add*: *True*)
  next
    case *False*

```
    with Suc have (S i, S j) ∈ 𝒜* by simp
    moreover from assms have (S j, S (j + 1)) ∈ 𝒜* by auto
    ultimately show ?thesis by simp
  qed
qed
```

A.3 Term Rewriting

If there is a non-root rewrite step between to terms s and t, there is an argument such that rewriting affects only this argument and leaves all the other arguments unchanged.

```
lemma nrrstep_args:
  assumes s ≥ϵ→ℛ t
  shows ∃f ss ts. s = Fun f ss ∧ t = Fun f ts ∧ |ss| = |ts|
    ∧ (∃j<|ss|. ss_j →ℛ ts_j ∧ (∀i<|ss|. i ≠ j ⟶ ss_i = ts_i))
proof −
  from assms obtain l r C σ where (l, r) ∈ ℛ and C ≠ □
    and s: s = C⟨lσ⟩ and t: t = C⟨rσ⟩ unfolding nrrstep_def' by best
  from ‹C ≠ □› obtain f ss1 D ss2 where C: C = More f ss1 D ss2
    by (induct C) auto
  have s = Fun f (ss1 @ D⟨lσ⟩ # ss2) (is _ = Fun f ?ss) by (simp add: s C)
  moreover have t = Fun f (ss1 @ D⟨rσ⟩ # ss2) (is _ = Fun f ?ts)
    by (simp add: t C)
  moreover have |?ss| = |?ts| by simp
  moreover
    have ∃j<|?ss|.  ?ss_j →ℛ ?ts_j ∧ (∀i<|?ss|. i ≠ j ⟶ ?ss_i = ?ts_i)
  proof −
    let ?j = |ss1|
    have ?j < |?ss| by simp
    moreover have ?ss_?j →ℛ ?ts_?j
    proof −
      from ‹(l, r) ∈ ℛ› have D⟨lσ⟩ →ℛ D⟨rσ⟩ by auto
      thus ?thesis by auto
    qed
    moreover have ∀i<|?ss|. i ≠ ?j ⟶ ?ss_i = ?ts_i
```

```
    (is ∀ i<|?ss|. _ ⟶ ?P i)
   proof (intro allI impI)
    fix i assume i < |?ss| and i ≠ ?j
    hence i < |ss1| ∨ i > |ss1| by auto
    thus ?P i
    proof
     assume i < |ss1| thus ?P i by (auto simp: nth_append)
    next
     assume i > |ss1| thus ?P i
      using ‹i < |?ss|› by (auto simp: nth_Cons' nth_append)
    qed
   qed
   ultimately show ?thesis by best
  qed
  ultimately show ?thesis by auto
qed
```

If a term is minimally nonterminating, then it is terminating with respect to non-root rewrite steps.

```
lemma Tinf_imp_SN_elt_nrrstep:
  assumes t ∈ 𝒯_ℛ^∞ shows SN^{≥ε}_{→ℛ}(t)
proof (rule ccontr)
  assume ¬ SN^{≥ε}_{→ℛ}(t)
  then obtain S where t = S 0 and nrseq: ∀ i. (S i) →^{≥ε}_ℛ (S (i+1)) by auto
  hence ∀ i. ∃ f ss ts. S i = Fun f ss ∧ S (i + 1) = Fun f ts ∧ |ss| = |ts|
    ∧ (∃ j<|ss|.  ss_j →_ℛ ts_j ∧ (∀ k<|ss|. k ≠ j ⟶ ss_k = ts_k))
      (is ∀ i. ∃ f ss ts. ?P i f ss ts) using nrrstep_args by fast
  from choice[OF this] obtain f where ∀ i. ∃ ss ts. ?P i (f i) ss ts by best
  from choice[OF this] obtain ss where ∀ i. ∃ ts. ?P i (f i) (ss i) ts by best
  from choice[OF this] obtain ts where P: ∀ i. ?P i (f i) (ss i) (ts i) by best
  moreover have ∀ i. f i = f (Suc i) ∧ ts i = ss (i + 1)
  proof
   fix i
   from P have S (i + 1) = Fun (f (i + 1)) (ss (i + 1)) by simp
   moreover from P have S (i + 1) = Fun (f i) (ts i) by blast+
```

```
  ultimately show f i = f (i + 1) ∧ ts i = ss (i + 1) by simp
qed
ultimately have ∀ i. ?P i (f i) (ss i) (ss (i + 1)) by auto
hence ∀ i. |ss i| = |ss (i + 1)|
  and ∀ i. ∃ j<|ss i|. (ss i)_j →_R (ss (i + 1))_j
    ∧ (∀ k<|ss i|. k ≠ j ⟶ (ss i)_k = (ss (i + 1))_k) by blast+
from choice[OF this(2)] obtain π
  where pi: ∀ i. π i<|ss i| ∧ (ss i)_(π i) →_R (ss (i + 1))_(π i)
    ∧ (∀ k<|ss i|. k ≠ π i ⟶ (ss i)_k = (ss (i + 1))_k) by blast+
hence seq: ∀ i. (ss i)_(π i) →_R (ss (i + 1))_(π i) by simp
have len: ∀ i. |ss i| = |ss 0|
proof
  fix i show |ss i| = |ss 0|
  proof (induct i)
    case 0 show ?case ..
  next
    case (Suc i) thus ?case using ‹∀ i. |ss i| = |ss (i + 1)|› by simp
  qed
qed
have ∀ i. π i < |ss 0|
proof
  fix i
  have |ss i| = |ss 0| using len by simp
  moreover from pi have π i < |ss i| by simp
  ultimately show π i < |ss 0| by simp
qed
hence range π ⊆ {i. i < |ss 0|} by auto
moreover have finite {i. i < |ss 0|} by simp
ultimately have finite (range π) by (rule finite_subset)
from finite_range[OF this] obtain j
  where j: ∀ i. ∃ k≥i. π k = j by (auto simp: INFM_nat_le)
from choice[OF this] obtain q where q: ∀ i. q i ≥ i ∧ π (q i) = j by auto
have j_len: ∀ i. j < |ss i|
proof
  fix i
  from j obtain k where k ≥ i and π k = j by best
  moreover from pi have π k < |ss k| by simp
```

ultimately have $j < |ss\ k|$ **by** *simp*
moreover from *len* **have** $|ss\ k| = |ss\ 0|$ **by** *simp*
moreover from *len* **have** $|ss\ i| = |ss\ 0|$ **by** *simp*
ultimately show $j < |ss\ i|$ **by** *simp*
qed
from *seq*
have *step*: $\forall\, i.\ (ss\ (q\ i))_{(\pi\ (q\ i))} \to_{\mathcal{R}} (ss\ ((q\ i)+1))_{(\pi\ (q\ i))}$ **by** *simp*
have *refl_seq*: $\forall\, i.\ (ss\ i)_j \to_{\mathcal{R}}^{=} (ss\ (i+1))_j$
proof
fix i **show** $(ss\ i)_j \to_{\mathcal{R}}^{=} (ss\ (i+1))_j$
proof (*cases* $\pi\ i = j$)
assume $\pi\ i = j$
moreover from *pi* **have** $(ss\ i)_{(\pi\ i)} \to_{\mathcal{R}} (ss\ (i+1))_{(\pi\ i)}$ **by** *simp*
ultimately show *?thesis* **by** *simp*
next
assume $\pi\ i \neq j$
moreover from $\langle \forall\, i.\ j < |ss\ i| \rangle$ **have** $j < |ss\ i|$ **..**
moreover from *pi*
have $j < |ss\ i| \wedge j \neq \pi\ i \longrightarrow (ss\ i)_j = (ss\ (i+1))_j$ **by** *simp*
ultimately show *?thesis* **by** *simp*
qed
qed
let $?s = \lambda i.\ (ss\ (shift\_by\ q\ i))_j$
have $\forall\, i.\ (ss\ i)_j \to_{\mathcal{R}}^{+} (ss\ ((q\ i)+1))_j$
proof
fix i
from q **have** $q\ i \geq i$ **by** *simp*
from *steps_imp_seq*[*OF refl_seq this*] **have** $(ss\ i)_j \to_{\mathcal{R}}^{*} (ss\ (q\ i))_j$ **by** *simp*
moreover from *step*
have $(ss\ (q\ i))_j \to_{\mathcal{R}} (ss\ ((q\ i)+1))_j$ **by** (*simp add*: q)
ultimately show $(ss\ i)_j \to_{\mathcal{R}}^{+} (ss\ ((q\ i)+1))_j$ **by** *simp*
qed
hence $\forall\, i.\ (?s\ i) \to_{\mathcal{R}}^{+} (?s\ (i+1))$ **by** *simp*
hence $\neg\ \mathsf{SN}_{\mathcal{R}}((?s\ 0))$ **using** *SN_elt_imp_SN_elt_trancl* **by** *best*
moreover have $?s\ 0 \lhd t$
proof $-$
from P **have** t: $t = Fun\ (f\ 0)\ (ss\ 0)$ **by** (*simp add*: $\langle t = S\ 0 \rangle$)

have $j < |ss\ 0|$ **using** *j_len* **by** *simp*
hence *?s 0* $\in$ *ss 0* **by** *simp*
thus *?thesis* **unfolding** *t* **..**
qed
moreover from *assms* **have** $\forall\, s \lhd t.\ \mathsf{SN}_{\mathcal{R}}(s)$ **unfolding** *Tinf_def* **by** *simp*
ultimately show *False* **by** *simp*
qed

A.4 Subterm Criterion

Projections are of type $\alpha \Rightarrow nat$. A projection π is lifted to terms as follows:

$\pi\ (Var\ x) = Var\ x$
$\pi\ (Fun\ f\ ts) = (\mathsf{if}\ \pi\ f < |ts|\ \mathsf{then}\ ts_{\pi\ f}\ \mathsf{else}\ Fun\ f\ ts)$

Reasoning about Projections.
Internally, we use *proj_term* π t to project the term t, using the projection π. But for readability, this function is dropped in the output.

Every subterm of a terminating term, is itself terminating.

lemma *subterm_preserves_SN*:
assumes *SN*: $\mathsf{SN}_{\mathcal{R}}(t)$ **and** *supt*: $(t,\ s) \in \rhd$
shows $\mathsf{SN}_{\mathcal{R}}(s)$
proof −
from *supt* **have** $t \trianglerighteq s$ **unfolding** *supt_def suptp_iff_supteqp_neq* **by** *simp*
thus *?thesis* **using** *SN_imp_SN_subt*[*OF SN*, *of s*] **by** *simp*
qed

If we have a rewrite step in a well-formed TRS, from a term s whose root is a constructor, to a term t, then after projecting, either, both terms are equal, or there is still a rewrite step between the projected terms.

lemma *rstep_proj_term*:
assumes $(\mathsf{root}(s),\ num\_args\ s) \notin \mathcal{D}_{\mathcal{R}}$ **and** $\mathsf{WF}(\mathcal{R})$ **and** $s \to_{\mathcal{R}} t$
shows $(\pi\ s) \to_{\mathcal{R}}^{=} (\pi\ t)$
proof −

```
from rstep_imp_Fun[OF ‹WF(R)› ‹s →_R t›]
  obtain f ss where s: s = Fun f ss by best
from ‹s →_R t› have (Fun f ss) →_R t by (simp add: s)
with ‹WF(R)› show ?thesis
proof (cases rule: rstep_cases_Fun')
  case (root ls r σ)
  hence (root(s), num_args s) ∈ D_R by (auto simp: s defined_def)
  with ‹(root(s), num_args s) ∉ D_R› show ?thesis ..
next
  case (nonroot i u)
  let ?ss1 = take i ss
  let ?ss2 = drop (Suc i) ss
  let ?C = More f ?ss1 □ ?ss2
  let ?ss = ?ss1 @ u # ?ss2
  show ?thesis
  proof (cases π f = i)
    case True
    have proj_s: π s = ss_i using True nonroot by (simp add: s)
    from nth_append_take[of i ss u] and ‹i < |ss|›
      have proj_t: π t = u unfolding nonroot using True by simp
    from ‹(ss_i) →_R u› show ?thesis unfolding proj_s proj_t ..
  next
    case False show ?thesis
    proof (cases π f < i)
      case True
      with nonroot have π f < |ss| by simp
      hence proj_s: π s = ss_(π f) by (simp add: s)
      from True and ‹π f < |ss|› have π f < |?ss1| by simp
      hence π t = ?ss1_(π f)
        unfolding nonroot
        using nth_append[of ?ss1 _ π f] by auto
      with nth_take[OF True, of ss]
        have proj_t: π t = ss_(π f) by simp
      show ?thesis unfolding proj_s proj_t by simp
    next
      case False
      with ‹π f ≠ i› have π f > i by simp
```

```
      show ?thesis
      proof (cases π f < |ss|)
        case False
        hence proj_s: π s = s unfolding s by simp
        from nonroot have |?ss| = |ss| by simp
        with False have ¬ π f < |?ss| by simp
        hence proj_t: π t = t unfolding nonroot by simp
        from ‹s →_R t› show ?thesis unfolding proj_s proj_t ..
      next
        case True
        hence π f ≤ |ss| by simp
        from ‹i < |ss|› have i ≤ |ss| by simp
        from nth_append_take_drop_is_nth_conv[OF ‹π f ≤ |ss|›
          ‹i ≤ |ss|› ‹π f ≠ i›]
          have ?ss_{π f} = ss_{π f} .
        hence proj_s: π s = ss_{π f}
          and proj_t: π t = ss_{π f}
          using ‹π f < |ss|› by (simp_all add: nonroot s)
        show ?thesis unfolding proj_s proj_t by simp
      qed
    qed
  qed
qed
```

If a term is terminating with respect to a TRS, then it is still terminating, when allowing proper subterm steps in addition to rewrite steps.

```
lemma SN_elt_rstep_imp_SN_elt_supt_union_rstep:
  assumes SN_R(t)
  shows SN_{(▷ ∪ →_R)}(t) (is SN_{?S}(t))
proof (rule ccontr)
  assume ¬ SN_{?S}(t)
  then obtain s where ini: s 0 = t and seq: ∀ i. (s i, s (i + 1)) ∈ ?S
    unfolding SN_defs by auto
  have SN: ∀ i. SN_R(s i)
  proof
```

```
    fix i show SN_R(s i)
    proof (induct i)
      case 0 show ?case using assms unfolding ⟨s 0 = t⟩ .
    next
      case (Suc i)
      from seq have (s i, s (i + 1)) ∈ ?S by simp
      thus ?case
      proof
        assume (s i, s (i + 1)) ∈ ▷
        from subterm_preserves_SN[OF Suc this] show ?thesis .
      next
        assume (s i) →_R (s (i + 1))
        from step_preserves_SN_elt[OF this Suc] show ?thesis .
      qed
    qed
  qed
  have no_seq: ¬ (∃ S. ∀ i. (S i, S (i + 1)) ∈ SN_rel_supt (→_R))
    using SN_SN_rel_supt_rstep unfolding SN_defs by auto
  have ∀ i. (s i, s (i + 1)) ∈ SN_rel_supt (→_R)
  proof
    fix i
    from SN have SN: SN_R((s i)) by simp
    from seq have (s i, s (i + 1)) ∈ ▷ ∪ →_R by simp
    thus (s i, s (i + 1)) ∈ SN_rel_supt (→_R)
      unfolding SN_rel_supt_def SN_rel_def using SN by auto
  qed
  with no_seq show False by auto
qed
```

We define the restriction of a TRS $\mathcal{R}$ to only those rules that are well-formed (that is, no variable as left-hand side and all variables in the right-hand side do also occur in the left-hand side) as follows:

$$wf\_rules\ \mathcal{R} \equiv \{(l,\ r) \mid (l,\ r) \in \mathcal{R} \wedge l \notin \mathcal{V} \wedge \mathcal{V}\mathsf{ar}(r) \subseteq \mathcal{V}\mathsf{ar}(l)\}$$

If there is a rewrite step from a terminating term, then the same step is still possible when considering just those rules of the TRS that satisfy the variable

conditions.

```
lemma SN_elt_imp_rstep_wf_rules:
  assumes SN_R(s) and s →_R t shows s →_{wf_rules R} t
using ⟨s →_R t⟩ proof
  fix l r C σ assume (l, r) ∈ R and s: s = C[lσ] and t: t = C[rσ]
  show ?thesis
  proof (cases (l, r) ∈ wf_rules R)
    case True thus ?thesis by (auto simp: s t)
  next
    case False
    with ⟨(l, r) ∈ R⟩ have is_var l ∨ (∃x. x ∈ Var(r) − Var(l))
      by (auto simp: wf_rules_def wf_rule_def)
    thus ?thesis
    proof
      assume is_var l
      then obtain x where l: l = Var x by auto
      let ?σ = {x/r}
      let ?S = λi. C[(Var x)(?σ^i)σ]
      have ?S 0 = s by (simp add: s l)
      moreover have ∀i. (?S i) →_R (?S (i + 1))
      proof
        fix i
        from rstep.ctxt[OF rstep.subst[OF rstep.id[OF ⟨(l, r) ∈ R⟩],
          of (?σ^i) ∘ σ], of C]
          show (?S i) →_R (?S (i + 1)) by (simp add: l subst_def)
      qed
      ultimately have ¬ SN_R(s) unfolding SN_defs by best
      with assms show ?thesis by simp
    next
      assume ∃x. x ∈ Var(r) − Var(l)
      then obtain x where x ∈ Var(r) − Var(l)
        and empty: Var(l) ∩ {x} = ∅ by auto
      hence r ⊵ Var x by (induct r) auto
      then obtain D where r: r = D[Var x] by (rule supteqp_ctxt_E)
      let ?σ = {x/l}
      let ?S = λi. C[((D?σ)^i)[l]σ]
```

```
    from subst_apply_id'[OF empty, of ?σ]
      have l: l?σ = l by simp
    have ?S 0 = s by (simp add: s)
    moreover have ∀ i. (?S i) →_R (?S (i + 1))
    proof
      fix i
      from rstepI[OF ⟨(l, r) ∈ R⟩, of _ (D?σ)^i ?σ]
        have ((D?σ)^i)[l] →_R (((D?σ)^i) ∘ ((D?σ)^(0 + 1)))[l?σ]
        unfolding r l subst_apply_term_ctxt_apply_distrib
        by (simp add: subst_def)
      from rstep.ctxt[OF rstep.subst[OF this], of C σ]
        show (?S i) →_R (?S (i + 1))
        unfolding ctxt_power_compose_distr[symmetric] by (simp add: l)
    qed
    ultimately have ¬ SN_R(s) unfolding SN_defs by best
    with assms show ?thesis by simp
  qed
 qed
qed
```

A rewrite sequence starting at a terminating term, is still possible when restricting to only those rules of the used TRS that satisfy the variable conditions.

```
lemma SN_elt_imp_rsteps_wf_rules:
 assumes s →*_R t and SN_R(s) shows s →*_(wf_rules R) t
using ⟨s →*_R t⟩ proof (induct)
 case base show ?case ..
next
 case (step u v)
 from steps_preserve_SN_elt[OF ⟨s →*_R u⟩ ⟨SN_R(s)⟩]
   have SN_R(u) .
 from SN_elt_imp_rstep_wf_rules[OF this ⟨u →_R v⟩]
   have u →_(wf_rules R) v .
 with ⟨s →*_(wf_rules R) u⟩ show ?case ..
qed
```

Consider a rewrite step from a terminating term s to some term t. When

projecting the terms, they either become equal, or there is still a rewrite step between the projected terms.

lemma *SN_elt_rstep_proj_term*:
  assumes $\mathsf{SN}_{\mathcal{R}}(s)$ **and** $(\mathsf{root}(s), \mathit{num\_args}\ s) \notin \mathcal{D}_{\mathcal{R}}$ **and** $s \to_{\mathcal{R}} t$
  shows $(\pi\ s) \to_{\mathcal{R}}^{=} (\pi\ t)$
proof −
  from *assms* **have** *undef*: $(\mathsf{root}(s), \mathit{num\_args}\ s) \notin \mathcal{D}_{\mathit{wf\_rules}\ \mathcal{R}}$
    unfolding *defined_def wf_rules_def* **by** *auto*
  from *SN_elt_imp_rstep_wf_rules*[*OF* ‹$\mathsf{SN}_{\mathcal{R}}(s)$› ‹$s \to_{\mathcal{R}} t$›]
    have $s \to_{\mathit{wf\_rules}\ \mathcal{R}} t$ **.**
  from *rstep_proj_term*[*OF undef wf_trs_wf_rules this*]
    show *?thesis* **by** *blast*
qed

Rewrite sequences, starting at terminating terms with constructors as roots, are preserved by projections.

lemma *SN_elt_rsteps_proj_term*:
  assumes $s \to_{\mathcal{R}}^{*} t$ **and** $\mathsf{SN}_{\mathcal{R}}(s)$ **and** $(\mathsf{root}(s), \mathit{num\_args}\ s) \notin \mathcal{D}_{\mathcal{R}}$
  shows $(\pi\ s) \to_{\mathcal{R}}^{*} (\pi\ t)$
proof −
  from *assms* **have** *undef*: $(\mathsf{root}(s), \mathit{num\_args}\ s) \notin \mathcal{D}_{\mathit{wf\_rules}\ \mathcal{R}}$
    unfolding *defined_def wf_rules_def* **by** *auto*
  from *SN_elt_imp_rsteps_wf_rules*[*OF* ‹$s \to_{\mathcal{R}}^{*} t$› ‹$\mathsf{SN}_{\mathcal{R}}(s)$›]
    have $s \to_{\mathit{wf\_rules}\ \mathcal{R}}^{*} t$ **.**
  from *rsteps_proj_term*[*OF this undef wf_trs_wf_rules*, *of* π]
    show *?thesis*
  proof (*induct*)
    case *base* **thus** *?case* **..**
  next
    case (*step u v*)
    with *rstep_wf_rules_subset*[*of* $\mathcal{R}$] **have** $u \to_{\mathcal{R}} v$ **by** *blast*
    with ‹$(\pi\ s) \to_{\mathcal{R}}^{*} u$› **show** *?case* **..**
  qed
qed

A.5 Signature Extensions

If there is a rewrite step between the two terms s and t, then cleaning either turns them equal, or there is still a rewrite step between the cleaned terms.

lemma *rstep_imp_clean_rstep_or_Id*:
assumes $\mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}$ **and** $s \rightarrow_{\mathcal{R}} t$
shows $[\![s]\!]_{\mathcal{F}} \rightarrow_{\mathcal{R}}^{=} [\![t]\!]_{\mathcal{F}}$
proof −
from ‹$s \rightarrow_{\mathcal{R}} t$› **obtain** $C\ \sigma\ l\ r$
where $(l, r) \in \mathcal{R}$ **and** s: $s = C[l\sigma]$ **and** t: $t = C[r\sigma]$ **..**
from $s\ t$ **show** *?thesis*
proof (*induct C arbitrary*: $s\ t$)
case *Hole*
let *?c* = $\lambda\sigma$. *case* σ *of Subst s* ⇒ *Subst* ($[\![\cdot]\!]_{\mathcal{F}} \circ s$)
from *clean_subst_apply_term*[*OF lhs_wf*[*OF* ‹$(l, r) \in \mathcal{R}$› ‹$\mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}$›]]
have s: $[\![s]\!]_{\mathcal{F}} = l($*?c* $\sigma)$ **by** (*cases* σ) (*simp_all add*: *Hole*)
from *clean_subst_apply_term*[*OF rhs_wf*[*OF* ‹$(l, r) \in \mathcal{R}$› ‹$\mathcal{F}(\mathcal{R}) \subseteq \mathcal{F}$›]]
have t: $[\![t]\!]_{\mathcal{F}} = r($*?c* $\sigma)$ **by** (*cases* σ) (*simp_all add*: *Hole*)
from ‹$(l, r) \in \mathcal{R}$› **show** *?case* **unfolding** $s\ t$ **by** *auto*
next
case (*More f ss1 D ss2*)
hence *IH*: $[\![D[l\sigma]]\!]_{\mathcal{F}} \rightarrow_{\mathcal{R}}^{=} [\![D[r\sigma]]\!]_{\mathcal{F}}$ **by** *simp*
show *?case*
proof (*cases* $(f, |ss1 \mathbin{@} ss2| + 1) \in \mathcal{F}$)
case *False*
with *More* **have** $[\![s]\!]_{\mathcal{F}} = z$ **by** *simp*
moreover from *False* **and** *More* **have** $[\![t]\!]_{\mathcal{F}} = z$ **by** *simp*
ultimately show *?thesis* **by** *auto*
next
case *True*
let *?s* = $[\![D[l\sigma]]\!]_{\mathcal{F}}$
let *?t* = $[\![D[r\sigma]]\!]_{\mathcal{F}}$
from *IH* **show** *?thesis*
proof
assume $(?s, ?t) \in Id$ **with** *True* **and** *More* **show** *?thesis* **by** *auto*
next

```
      let ?C = More f (map [[·]]_F ss1) □ (map [[·]]_F ss2)
      assume ?s →_R ?t
      from rstep.ctxt[OF this, of ?C] and True
        have [[s]]_F →_R [[t]]_F by (simp add: More)
      thus ?thesis ..
    qed
  qed
 qed
qed
```

Cleaning preserves rewrite sequences.

```
lemma rsteps_imp_clean_rsteps:
 assumes F(R) ⊆ F and s →*_R t
 shows [[s]]_F →*_R [[t]]_F
using ⟨s →*_R t⟩ proof (induct rule: rtrancl_induct)
 case base show ?case by simp
next
 case (step u v)
 from rstep_imp_clean_rstep_or_Id[OF ⟨F(R) ⊆ F⟩ ⟨u →_R v⟩]
   have [[u]]_F →=_R [[v]]_F .
 thus ?case using step by auto
qed
```

Appendix B

Publications

For completeness and since not all of my publications that originated during my PhD studies are directly connected to this thesis (and hence not referenced), here is a list of all my publications (in order of appearance).

- Christian Sternagel and Aart Middeldorp. Root-Labeling. In Andrei Voronkov, editor, *Rewriting Techniques and Applications, RTA 2008*, volume 5117 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2008.
  DOI 10.1007/978-3-540-70590-1_23.

- Christian Sternagel, René Thiemann, Sarah Winkler, and Harald Zankl. CeTA - A Tool for Certified Termination Analysis. In *Workshop on Termination, WST 2009*, 2009.

- Christian Sternagel and René Thiemann. Loops under Strategies. In Ralf Treinen, editor, *Rewriting Techniques and Applications, RTA 2009*, volume 5595 of *Lecture Notes in Computer Science*, pages 17–31. Springer, 2009.
  DOI 10.1007/978-3-642-02348-4_2.

- Martin Korp, Christian Sternagel, Harald Zankl, and Aart Middeldorp. Tyrolean Termination Tool 2. In Ralf Treinen, editor, *Rewriting Techniques and Applications, RTA 2009*, volume 5595 of *Lecture Notes in Computer Science*, pages 295–304. Springer, 2009.
  DOI 10.1007/978-3-642-02348-4_21.

- Harald Zankl, Christian Sternagel, and Aart Middeldorp. Transforming SAT into Termination of Rewriting. In Moreno Falaschi, editor, *Workshop on Functional and (Constraint) Logic Programming, WFLP 2008*, volume 246 of *Electronic Notes in Theoretical Computer Science*, pages 199–214. Elsevier B.V., 2009.
  DOI `10.1016/j.entcs.2009.07.023`.

- René Thiemann and Christian Sternagel. Certification of Termination Proofs using CeTA. In Stefan Berghofer et al., editors, *Theorem Proving in Higher Order Logics, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468. Springer, 2009.
  DOI `10.1007/978-3-642-03359-9_31`.

- Harald Zankl, Christian Sternagel, Dieter Hofbauer, and Aart Middeldorp. Finding and Certifying Loops. In Jan van Leeuwen et al., editors, *Theory and Practice of Computer Science, SOFSEM 2010*, volume 5901 of *Lecture Notes in Computer Science*, pages 755–766. Springer, 2010.
  DOI `10.1007/978-3-642-11266-9_63`.

- Christian Sternagel and René Thiemann. Abstract Rewriting. In Gerwin Klein et al., editors, *The Archive of Formal Proofs*.
  `http://afp.sf.net/entries/Abstract-Rewriting.shtml`, June 2010. Formal proof development.

- Christian Sternagel and René Thiemann. Executable Matrix Operations on Matrices of Arbitrary Dimensions. In Gerwin Klein et al., editors, *The Archive of Formal Proofs*.
  `http://afp.sf.net/entries/Matrix.shtml`, June 2010. Formal proof development.

- Christian Sternagel and René Thiemann. Certified Subterm Criterion and Certified Usable Rules. In Christopher Lynch, editor, *Rewriting Techniques and Applications, RTA 2010*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 325–340. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2010.
  DOI `dx.doi.org/10.4230/LIPIcs.RTA.2010.325`.

- René Thiemann, Jürgen Giesl, Peter Schneider-Kamp, and Christian Sternagel. Loops under Strategies ... Continued. In *International Work-*

shop on Strategies in Rewriting, Proving, and Programming, IWS 2010, 2010.

- Christian Sternagel and René Thiemann. Certification extends Termination Techniques. In *Workshop on Termination, WST 2010*, 2010.
- Christian Sternagel and René Thiemann. Signature Extensions Preserve Termination - An Alternative Proof via Dependency Pairs. In Anuj Dawar and Helmut Veith, editors, *Computer Science Logic, CSL 2010*, volume 6247 of *Lecture Notes in Computer Science*, pages 514–528. Springer, 2010. to appear

Bibliography

[1] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000. DOI 10.1016/S0304-3975(99)00207-8.

[2] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, August 1999. ISBN 0-521-77920-0.

[3] Marc Bezem and Thierry Coquand. Newman's lemma - a case study in proof automation and geometric logic. *Bulletin of the EATCS*, 79:86–100, 2003.

[4] Frédéric Blanqui and Adam Koprowski. Automated verification of termination certificates. Technical Report 6949, INRIA, June 2009. http://www-rocq.inria.fr/~blanqui/papers/rr09color.pdf.

[5] Frédéric Blanqui, Solange Coupte-Grimal, William Delobel, Sébastien Hinderer, and Adam Koprowski. CoLoR, a Coq library on rewriting and termination. In Alfons Geser and Harald Søndergaard, editors, *Proceedings of the 8th International Workshop on Termination, WST 2006*, pages 69–73, 2006.

[6] Évelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Certification of automated termination proofs. In Boris Konev and Frank Wolter, editors, *Frontiers of Combining Systems, FroCos 2007*, volume 4720 of *Lecture Notes in Computer Science*, pages 148–162. Springer, 2007. DOI 10.1007/978-3-540-74621-8_10.

[7] Évelyne Contejean, Andrei Paskevich, Xavier Urbain, Pierre Courtieu, Olivier Pons, and Julien Forest. A3PAT, an approach for certified automated termination proofs. In John Patrick Gallagher and Janis

Voigtländer, editors, *ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010*, pages 63–72. ACM New York, NY, USA, 2010.
DOI 10.1145/1706356.1706370.

[8] Pierre Courtieu, Julien Forest, and Xavier Urbain. Certifying a termination criterion based on graphs, without graphs. In Otmane Ait Mohamed, César Muñoz, and Sofiéne Tahar, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2008*, volume 5170 of *Lecture Notes in Computer Science*, pages 182–198. Springer, 2008.
DOI 10.1007/978-3-540-71067-7_17.

[9] Nachum Dershowitz. Termination dependencies. In Albert Rubio, editor, *Proceedings of the 6th International Workshop on Termination, WST 2003*, pages 27–30, 2003.

[10] Jörg Endrullis. Jambox, 2005.
http://joerg.endrullis.de.

[11] André Luiz Galdino and Mauricio Ayala-Rincón. A formalization of Newman's and Yokouchi's lemmas in a higher-order language. *Journal of Formalized Reasoning*, 1(1):39–50, 2008.

[12] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In Franz Baader and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, LPAR 2004*, volume 3452 of *Lecture Notes in Computer Science*, pages 301–331. Springer, 2005.
DOI 10.1007/b106931.

[13] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. Proving and disproving termination of higher-order functions. In Bernhard Gramlich, editor, *Frontiers of Combining Systems, FroCos 2005*, volume 3717 of *Lecture Notes in Computer Science*, pages 216–231. Springer, 2005.
DOI 10.1007/11559306_12.

[14] Jürgen Giesl, René Thiemann, Peter Schneider-Kamp, and Stephan Falke. Mechanizing and improving dependency pairs. *Journal of Automated Reasoning*, 37(3):155–203, 2006.
DOI 10.1007/s10817-006-9057-7.

[15] Florian Haftmann. *Code generation from Isabelle/HOL theories*, June 2010.

[16] Nao Hirokawa and Aart Middeldorp. Automating the dependency pair method. In Franz Baader, editor, *Automated Deduction, CADE 2003*, pages 32–46. Springer, 2003.
DOI 10.1145/978-3-540-45085-6_4.

[17] Nao Hirokawa and Aart Middeldorp. Dependency pairs revisited. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications, RTA 2004*, volume 3091 of *Lecture Notes in Computer Science*, pages 249–268. Springer, 2004.
DOI 10.1007/b98160.

[18] Nao Hirokawa and Aart Middeldorp. Tyrolean termination tool: Techniques and features. *Information and Computation*, 205(4):474–511, 2007.
DOI 10.1016/j.ic.2006.08.010.

[19] Gérard P. Huet. Residual theory in λ-calculus: A formal development. *Journal of Functional Programming*, 4(3):371–394, 1994.
DOI 10.1017/S0956796800001106.

[20] Adam Koprowski. *Termination of Rewriting and Its Certification.* PhD thesis, Technische Universiteit Eindhoven, 2008.

[21] Adam Koprowski. Coq formalization of the higher-order recursive path ordering. *Applicable Algebra in Engineering, Communication and Computing*, 20(5-6):379–425, 2009.
DOI 10.1007/s00200-009-0105-5.

[22] Adam Koprowski and Johannes Waldmann. Arctic termination ... below zero. In Andrei Voronkov, editor, *Rewriting Techniques and Applications, RTA 2008*, pages 202–216. Springer, 2008.
DOI 10.1007/978-3-540-70590-1_14.

[23] Adam Koprowski and Hans Zantema. Certification of proving termination of term rewriting by matrix interpretations. In Viliam Geffert, Juhani Karhumäki, Alberto Bertoni, Bart Preneel, Pavol Návrat, and Mária Bieliková, editors, *Current Trends in Theory and Practice of Computer*

Science, SOFSEM 2008, volume 4910 of *Lecture Notes in Computer Science*, pages 328–339. Springer, 2008.
DOI 10.1007/978-3-540-77566-9.

[24] Alexander Krauss. Certified size-change termination. In Frank Pfenning, editor, *Automated Deduction, CADE 2007*, volume 4603 of *Lecture Notes in Computer Science*, pages 460–475. Springer, 2007.
DOI 10.1007/978-3-540-73595-3_34.

[25] Alexander Krauss. *Automating Recursive Definitions and Termination Proofs in Higher-Order Logic*. PhD thesis, Technische Universität München, Institut für Informatik, 2009.

[26] Donald MacKenzie. *Mechanizing Proof*. MIT Press, March 2004. ISBN 978-0-262-63295-9. Paperback.

[27] Aart Middeldorp. *Modular Properties of Term Rewriting Systems*. PhD thesis, Vrije Universiteit, Amsterdam, 1990.

[28] Tobias Nipkow. Structured proofs in Isar/HOL. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, TYPES 2002*, volume 2646 of *Lecture Notes in Computer Science*, pages 259–278. Springer, 2003.
DOI 10.1007/3-540-39185-1_15.

[29] Tobias Nipkow, Lawrence Charles Paulson, and Makarius Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. ISBN 978-3-540-43376-7. It is strongly recommended to download its updated version, which is part of the documentation of the latest Isabelle release.
DOI 10.1007/3-540-45949-9.

[30] Enno Ohlebusch. A simple proof of sufficient conditions for the termination of the disjoint union of term rewriting systems. *Bulletin of the EATCS*, 50:223–228, 1993.

[31] Lawrence Charles Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5(3):363–397, September 1989.
DOI 10.1007/BF00248324.

[32] Ole Rasmussen. The Church-Rosser theorem in Isabelle: A proof porting experiment. Technical Report UCAM-CL-TR-364, University of Cambridge, March 1995.

[33] José-Luis Ruiz-Reina, José-Antonio Alonso, María-José Hidalgo, and Francisco-Jesús Martín-Mateos. Formalizing rewriting in the ACL2 theorem prover. In John A. Campbell and Eugenio Roanes-Lozano, editors, *Artificial Intelligence and Symbolic Computation, AISC 2000*, volume 1930 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2000. DOI 10.1007/3-540-44990-6_7.

[34] Natarajan Shankar. A mechanical proof of the Church-Rosser theorem. *Journal of the ACM*, 35(3):475–522, 1988. DOI 10.1145/44483.44484.

[35] Christian Sternagel and Aart Middeldorp. Root-Labeling. In Andrei Voronkov, editor, *Rewriting Techniques and Applications, RTA 2008*, volume 5117 of *Lecture Notes in Computer Science*, pages 336–350. Springer, 2008. DOI 10.1007/978-3-540-70590-1_23.

[36] Christian Sternagel and René Thiemann. Abstract Rewriting. In Gerwin Klein, Tobias Nipkow, and Lawrence Charles Paulson, editors, *The Archive of Formal Proofs*. http://afp.sf.net/entries/Abstract-Rewriting.shtml, June 2010. Formal proof development.

[37] Christian Sternagel and René Thiemann. Executable Matrix Operations on Matrices of Arbitrary Dimensions. In Gerwin Klein, Tobias Nipkow, and Lawrence Charles Paulson, editors, *The Archive of Formal Proofs*. http://afp.sf.net/entries/Matrix.shtml, June 2010. Formal proof development.

[38] Christian Sternagel and René Thiemann. Certified subterm criterion and certified usable rules. In Christopher Lynch, editor, *Rewriting Techniques and Applications, RTA 2010*, volume 6 of *Leibniz International Proceedings in Informatics*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010. DOI 10.4230/LIPIcs.RTA.2010.325.

[39] Christian Sternagel and René Thiemann. Signature extensions preserve termination - an alternative proof via dependency pairs. In Anuj Dawar

and Helmut Veith, editors, *EACSL Annual Conference on Computer Science Logic, CSL 2010*, volume 6247 of *Lecture Notes in Computer Science*, pages 514–528. Springer, 2010. to appear.

[40] Christian Sternagel, René Thiemann, Sarah Winkler, and Harald Zankl. CeTA - a tool for certified termination analysis. In *Proceedings of the 10th International Workshop on Termination, WST 2009*, 2009.

[41] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003. ISBN 0-521-39115-6.

[42] René Thiemann. *The DP Framework for Proving Termination of Term Rewriting*. PhD thesis, Rheinisch-Westfälische Technische Hochschule Aachen, Department of Computer Science, 2007.

[43] René Thiemann and Christian Sternagel. Certification of termination proofs using CeTA. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, TPHOLs 2009*, volume 5674 of *Lecture Notes in Computer Science*, pages 452–468. Springer, 2009.
DOI 10.1007/978-3-642-03359-9_31.

[44] Hélène Touzet. A complex example of a simplifying rewrite system. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *International Colloquium on Automata, Languages and Programming, ICALP 1998*, volume 1443 of *Lecture Notes in Computer Science*, pages 507–517. Springer, 1998.
DOI 10.1007/BFb0055035.

[45] Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, December 1937.
DOI 10.1112/plms/s2-42.1.230.

[46] Xavier Urbain. Modular and incremental automated termination proofs. *Journal of Automated Reasoning*, 32(4):315–355, 2004.
DOI 10.1007/BF03177743.

[47] Johannes Waldmann. Matchbox: A tool for match-bounded string rewriting. In Vincent van Oostrom, editor, *Rewriting Techniques and Applications, RTA 2004*, volume 3091 of *Lecture Notes in Computer Science*, pages 85–94. Springer, 2004.
DOI 10.1007/b98160.

[48] Makarius Wenzel. *Isabelle/Isar - A Versatile Environment for Human-Readable Formal Proof Documents*. PhD thesis, Technische Universität München, Institut für Informatik, 2002.

[49] Makarius Wenzel. *The Isabelle/Isar Implementation*, December 2009.

[50] Makarius Wenzel. *The Isabelle/Isar Reference Manual*, December 2009.

[51] Harald Zankl, Christian Sternagel, Dieter Hofbauer, and Aart Middeldorp. Finding and certifying loops. In Jan van Leeuwen, Anca Muscholl, David Peleg, Jaroslav Pokorný, and Bernhard Rumpe, editors, *Current Trends in Theory and Practice of Computer Science, SOFSEM 2010*, volume 5901 of *Lecture Notes in Computer Science*, pages 755–766. Springer, 2010.
DOI 10.1007/978-3-642-11266-9_63.

[52] Hans Zantema. Termination of term rewriting by semantic labelling. *Fundamenta Informaticae*, 24(1/2):89–105, 1995.

Index

Page numbers of definitions (of concepts) and proofs (of lemmas) are printed **bold**.