Infinitary Rewriting in Coq

Martijn Vermaat

VU UNIVERSITY AMSTERDAM

Infinitary Rewriting in Coq

THESIS

submitted in partial fulfillment of the requirements for
the degree of Master of Science by research at
the VU University Amsterdam

by Martijn Vermaat

August 2010

SUPERVISOR
dr. R.D.A. Hendriks

SECOND READER
dr. R.C. de Vrijer

VU University *Amsterdam*

Typeset using LaTeX by Leslie Lamport, based on TeX by Donald Knuth.

Creative Commons Attribution logo by Sebastian Pipping.

Oiseau d'infini cover by Bastiaan Terhorst, `http://perceptor.nl`.

Abstract

This thesis describes our formalisation of the basic notions from the theory of infinitary term rewriting in the Coq proof assistant. Infinitary term rewriting is a generalisation of term rewriting where we allow terms to be infinitely deep and admit rewrite sequences of transfinite length. Coq is an interactive proof assistant based on constructive type theory with inductive types. The main contribution of our formalisation is an inductive definition of transfinite rewrite sequences, based on tree ordinals.

The ordinal numbers are represented by tree ordinals, following the construction of an order on tree ordinals by Hancock (2008). Infinite terms are defined by coinduction. A novel representation for rewrite sequences of ordinal length is defined, based on the inductive structure of the tree ordinals. The order on ordinals is then lifted to an embedding relation on rewrite sequences. We discuss an application of our formalisation in the verification of a counterexample to unique normal forms in weakly orthogonal infinitary TRSs, introduced by Endrullis, Grabmayer, Hendriks, Klop, and van Oostrom (2010).

Preface

This is my Master thesis in Computer Science, specialisation *Formal Methods and Software Verification*, submitted at the *VU University Amsterdam* (or just *Vrije Universiteit* if you like).

It describes the formalisation of some of the *infinitary rewriting* theory in the Coq proof assistant. Together, the formalisation and thesis make for a 36 ECTS project, roughly equivalent to six months of work, concluding my Master study.

So I would like to say to myself, *welcome to the real world!* But not before some much deserved *thank-you*'s (and the remainder of this thesis).

Acknowledgements

First of all, I would like to thank my supervisor, Dimitri Hendriks. Almost a year ago, he coined the title of this thesis, and has since spent many hours fixing my Coq code, or otherwise 'supervising' me. The formalisation of infinite terms described in Section 3.2, for example, is in large part due to Dimitri.

I also thank Roel de Vrijer, as second reader of my thesis, but also as the one who introduced me to the worlds of formal methods and logic—two central themes in my Master study.

Though not directly in this final project, Femke van Raamsdonk was involved in many of my curricular activities, be it as lecturer, supervisor, or employer, but always in a friendly and colleague-like way. Thanks!

The work described in this thesis was also at least partially enabled by the following people. So thank you. Peter Hancock for his notes on tree ordinals and

further personal communication. Vincent van Oostrom for a fruitful morning session and useful comments on the embedding of rewrite sequences. Clemens Grabmayer for elaborating on the infinitary unique normal form property. Adam Chlipala for his recursive vector type and other (sometimes extremely fast) answers on the Coq mailing list. Bruno Barras for a hint on proving a lemma on ordinals, through that same mailing list. Matthieu Sozeau for assisting me on problems with a Coq tactic. And of course Jörg Endrullis for joining Dimitri and me on our coffee/chocolate/tea breaks, but also for many insightful comments on my project.

Finally, I thank Bastiaan Terhorst for *Oiseau d'infini* guarding the printed version of this thesis.

Martijn Vermaat
Amsterdam, August 2010

Contents

Chapter 1

Introduction

This thesis describes a formalisation of the theory of infinitary term rewriting in the Coq proof assistant. The foundation of Coq is a constructive type theory with inductive types, whereas infinitary term rewriting, building on the theory of finitary rewriting, uses notions from topology, set theory, and analysis, but not necessarily in a constructive way.

The central question we aim to answer in this thesis is whether the traditional notions from infinitary term rewriting can be translated to Coq in such a way that the resulting definitions are natural for the Coq system. Of course, for such a translation to be satisfactory, it should preserve the semantics of the original notions.

In the remainder we may simply write ‘rewriting’ instead of ‘infinitary rewriting’ and ‘term’ instead of ‘infinite term’.

Although this text contains a fair amount of Coq code, it is not our intention to completely list a development ready for compiling. Rather, the included code fragments are thought to be the most interesting ones for the purpose of discussion of our development. In fact, many of the code listings are simplified and/or typographically enhanced to a form beyond of what the Coq compiler will accept. Furthermore, lemmas are stated without proof. The reader is invited to study the full source code, with proofs, which is available at `http://martijn.vermaat.name/master-project/`.

Infinitary Rewriting

The theory of *(finitary) rewriting* is concerned with the repeated transformation of objects by discrete steps following a predefined set of rules. Such a set of rules can be understood as implementing a programming language if we take programs as the objects to be transformed. Indeed, *term rewriting* is the foundational model of functional programming. Other examples of rewriting can be found in the transformation of braids (Melliès, 1995) and the λ-calculus. As a matter of fact, λ-calculus is of prime importance, both as a model of computation and as a logical framework.

Infinitary rewriting generalises finitary rewriting by considering infinitely large objects and series of transfinitely many transformation steps. One could question the validity of this generalisation, especially in the context of mechanical formalisation with which this thesis is concerned. After all, the word 'mechanical' implies finite restrictions on the amounts of space and time we can use.

However, mathematicians (and computer scientists for that matter) have long had ample reason to include the infinite in their work. In *The Quadrature of the Parabola*, Archimedes considers the infinite summation

$$1 + \frac{1}{4} + \frac{1}{16} + \frac{1}{64} + \dots$$

in his proof that the area of a parabolic segment is $4/3$ that of a certain inscribed triangle. Of course we cannot carry out the infinite computation to arrive at the outcome $4/3$, but we can represent it in finite space and manipulate this representation in finite time to deduce its outcome.

As another example to motivate the study of infinite objects, consider the simple Haskell program

```
f 0   where   f n = n : f (n + 1)
```

that defines the infinite stream of natural numbers. We can inspect the stream at any position, but by Haskell's lazy evaluation the stream is never fully computed. Again, the represented object takes an infinite amount of space to store and an infinite amount of time to compute, yet we can perfectly reason about it in finite space and time.

The theory of infinitary term rewriting is formally introduced in Chapter 2. In

that chapter, we define precisely which infinite objects are allowed and what we understand by transfinite sequences of rewrite steps.

Mechanical Formalisation

The translation of infinitary rewriting to Coq is an example of the mechanical formalisation of a mathematical theory. By 'mechanical' we mean that the formalisation must be in a form that can be manipulated by a machine. Definitions on paper or ideas in one's head do not qualify as mechanical formalisations. Furthermore, we want the formalisation to represent the theory's semantics.

The promises of mechanical formalisations include:

Confidence Proof-checkers can verify the correctness of proofs in a rigorous way. If we trust the implementation of the checker (which is usually kept as small as possible) and its execution on a computer, we can be confident that a verified proof is correct, even if we do not fully comprehend it.

Automation Tedious work, whether computationally hard or just boring, is often better suited to computers. They are faster than humans and precise.

Intuition Proofs on paper typically abstract away from seemingly uninteresting details. This is often a good thing, but sometimes the level of detail of a mechanical proof gives us that extra bit of insight to fully understand its workings. Another way of gaining intuition is by building tools on top of the formalisation, e.g. providing graphical representations of definitions.

Availability Formalised theories may be searched and browsed semantically using a computer instead of just syntactically. Furthermore, the internet provides us with excellent methods to share and copy these formalisations globally.

In 1976, the four colour theorem[1] was proven (Appel and Haken, 1976). The proof used a computer program to show that a particular set of 1,936 maps satisfy a certain property by exhaustive case analysis. This is an example of automation by translation to a computer. The correctness of the proof, however, remained debatable because this case analysis by computer was impossible to perform or verify by hand.

[1] 'Four colours suffice to colour any map such that no two neighbouring countries have the same colour.' Posed by Francis Guthrie in 1852.

Even though a simpler proof was published by Robertson, Sanders, Seymour, and Thomas (1997), it was not until Gonthier (2005) formally verified the entire proof, including the computer program part, that all remaining doubts were dispelled. This formalisation thus helped gain confidence in the validity of the theorem.

Large-scale mechanical formalisation of mathematics goes back to the 1960's with de Bruijn's AUTOMATH project (Nederpelt, Geuvers, and de Vrijer, 1994). A complete text book on analysis (Landau, 1965) was formalised and verified for correctness, but the system never gained widespread use.

Many formalisation efforts, using many different systems, have since been undertaken. A list of formally verified proofs for 100 mathematical theorems is maintained by Wiedijk (2008). Ongoing work is the FLYSPECK project (Hales, Harrison, McLaughlin, Nipkow, Obua, and Zumkeller, 2009) with as goal a formal proof of the Kepler conjecture,[1] expected to take up to 20 work-years to complete.

Outline

In Chapter 2 we introduce ordinal numbers and the theory of infinitary term rewriting. This is mostly a recapitulation of Terese (2003), included for self-containment, and can be seen as preliminaries for the later chapters.

The purpose of Chapter 3 is to present our formalisation of infinitary term rewriting in the COQ proof assistant. We start with a short introduction to COQ and then review the main parts of our development.

Our formalisation was used to prove that in infinitary rewriting, weak orthogonality does not imply uniqueness of normal forms. This application is discussed in Chapter 4.

Finally, in Chapter 5 we discuss our results and draw conclusions.

We include a short introduction to COQ in Appendix A.

[1] 'The most efficient packing of oranges is in a pyramid.' Posed by Johannes Kepler in 1611.

Chapter 2

Infinitary Term Rewriting

Before formally introducing term rewriting, we give an example of an infinitary TRS taken from Klop and de Vrijer (2005) and trust the reader to pick up the general ideas.

Let A, B, and P be symbols with arities 0, 1, and 2, respectively. We consider the system consisting of the following rule:

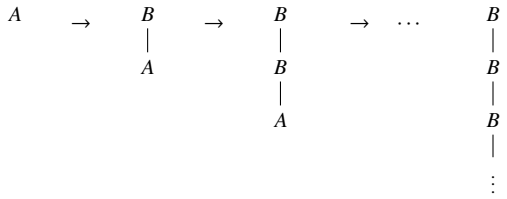
$$A \to B(A)$$

The term A rewrites to $B(B(B(A)))$, written $B^3(A)$, in 3 steps, or more generally to $B^n(A)$ in n steps for every $n \in \mathbb{N}$.

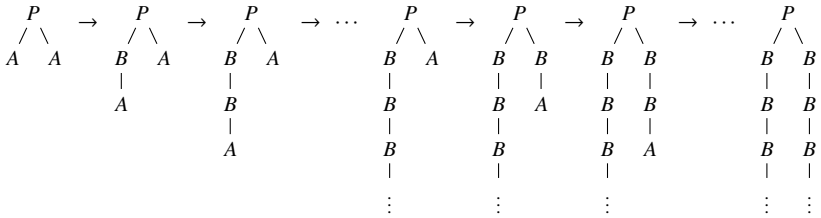
$$A \quad \to \quad \begin{array}{c} B \\ | \\ A \end{array} \quad \to \quad \begin{array}{c} B \\ | \\ B \\ | \\ A \end{array} \quad \to \quad \begin{array}{c} B \\ | \\ B \\ | \\ B \\ | \\ A \end{array}$$

In the finitary setting, these are the only rewrite sequences from A and all of them allow an additional step. Hence, A has no normal form.

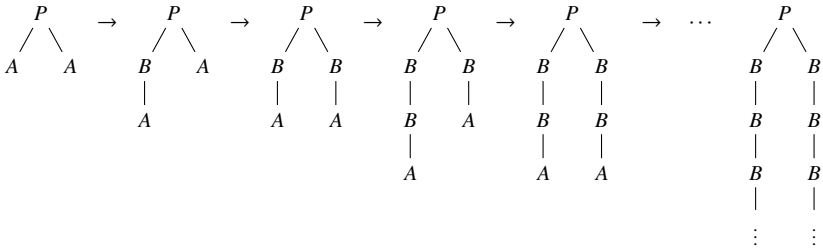
The limit of these rewrite sequences, although non-existent in finitary rewriting, is intuitively well-defined. It is reached in ω many rewrite steps and we denote it by B^ω. Infinitary rewriting allows (i) infinite terms such as B^ω and (ii) rewrite sequences of transfinite length such as $A \to^\omega B^\omega$.



The word ‘transfinite’ hints that there are also rewrite sequences of length $> \omega$. Consider for example the term $P(A, A)$. We can rewrite this term to $P(B^\omega, A)$ in ω many steps, and rewrite $P(B^\omega, A)$ to $P(B^\omega, B^\omega)$ in another ω many steps. Therefore we have $P(A, A) \to^{\omega\times 2} P(B^\omega, B^\omega)$.



However, $P(B^\omega, B^\omega)$ can also be reached from $P(A, A)$ in ω many steps, alternating the steps from the two ω-step rewrite sequences.



This observation is generalised in the Compression Lemma (page 16).

In the following sections, we introduce the ordinal numbers and a representation for them known as tree ordinals, and we present the basic notions from the theory of term rewriting as required in the following chapters.

2.1 Ordinal Numbers

Ordinal numbers (Cantor, 1915), or ordinals for short, are an extension of the natural numbers with transfinite objects. Indeed, the finite ordinals are just the

natural numbers. The smallest infinite ordinal is called ω and following ω we have $\omega + 1$, $\omega + 2$, ..., $\omega \times 2$. Then there are the ordinals $\omega \times 2 + 1$, $\omega \times 2 + 2$, ..., $\omega \times 3$. Some other (still relatively small) ordinals are:

$$\omega^2 \qquad \omega^\omega \qquad \omega^{\omega^2} \qquad \omega^{\omega^\omega} \qquad \omega^{\omega^{\omega^{\cdot^{\cdot^{\cdot}}}}} = \epsilon_0$$

Note that this is all merely notation, we have not yet defined a representation for ordinals or what $+$, $\times$, and superscripts are.

In set theory, ordinals are usually represented by hereditarily transitive sets. Zero corresponds to the empty set $\varnothing$, one to the singleton $\{\varnothing\}$ and so on, and ω is represented by $\{\varnothing, \{\varnothing\}, \{\varnothing, \{\varnothing\}\}, \ldots\}$. Now $\in$ constitutes a well-founded total order on the ordinals.

We abbreviate $\alpha \cup \{\alpha\}$ by α^+ and say that α is a *successor ordinal* if $\alpha = \beta^+$ for some ordinal β. If α is not a successor ordinal and $\alpha \neq \varnothing$, it is called a *limit ordinal*. Hence, an ordinal can be either zero, a successor ordinal, or a limit ordinal.

Examples of successor ordinals are 4, $\omega + 7$, and $\omega^{\omega \times 2} + 1$. Examples of limit ordinals are ω and $\omega \times 3$. Henceforth we use $\alpha, \beta, \gamma, \lambda$ to denote ordinals where λ always denotes a limit ordinal.

One can do arithmetic on ordinals much like we do arithmetics on natural numbers. For example, addition can be defined by recursion on the right argument:

$$\begin{aligned} \alpha + 0 &= \alpha \\ \alpha + \beta^+ &= (\alpha + \beta)^+ \\ \alpha + \lambda &= \bigcup \{\alpha + \gamma \mid \gamma \in \lambda\} \end{aligned}$$

2.1.1 Tree Ordinals

The tree ordinals (Dennis-Jones and Wainer, 1984) are a representation of the countable ordinals as countably branching well-founded trees. Their inductive definition uses constructors 0 (zero), $^+$ (successor), and $\sqcup$ (limit).

Definition 1. The set of *tree ordinals* $\mathcal{TO}$ is defined by induction:

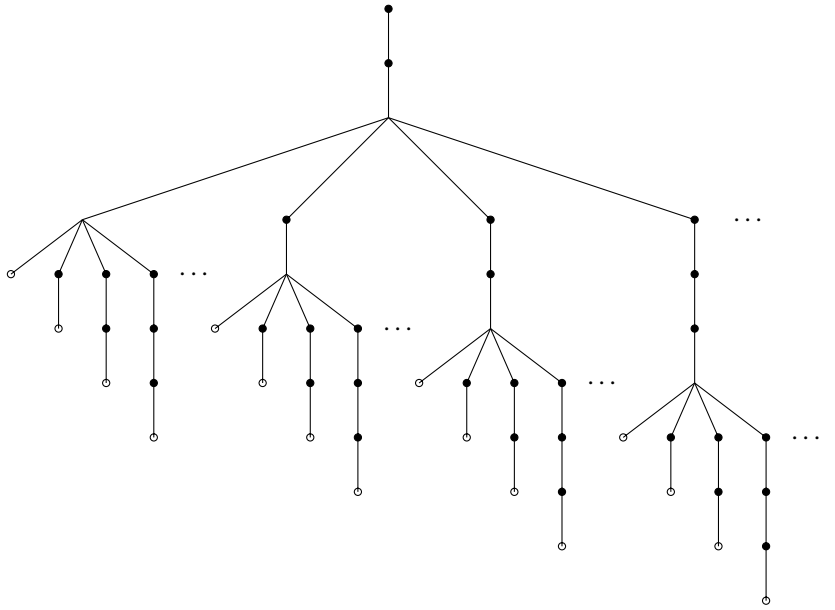


FIGURE 2.1. A representation of $\omega \times 2 + 2$. Branching points, filled dots, and open dots denote $\sqcup$, $^+$, and 0 constructors, respectively.

I. $0 \in \mathcal{TO}$.
II. If $\alpha \in \mathcal{TO}$, then $\alpha^+ \in \mathcal{TO}$.
III. If $\alpha_i \in \mathcal{TO}$ for all $i \in \mathbb{N}$, then $\sqcup_i \alpha_i \in \mathcal{TO}$. *

The $\sqcup$ constructor has type $(\mathbb{N} \to \mathcal{TO}) \to \mathcal{TO}$. For our convenience we write $\sqcup_i \cdots i \cdots$ instead of $\sqcup(\lambda i. \cdots i \cdots)$. Sometimes we explicitly enumerate the argument, writing for example $\sqcup(\alpha_1, \alpha_2, \alpha_3, \ldots)$.

Now zero is represented by 0, a successor ordinal $\alpha + 1$ is represented by α^+, and a limit ordinal λ is represented by $\sqcup_i \alpha_i$ if λ is the least upper bound of the sequence $\alpha_1, \alpha_2, \alpha_3, \ldots$. As an example of a tree ordinal representation, Figure 2.1 visualises $\omega \times 2 + 2$ as a tree ordinal.

Some ordinals have no unique representation as tree ordinal. Consider for example the limit ordinals $\sqcup_i i + 3$ and $\sqcup_i i \times 2$. Both are representations of ω and a meaningful order relation would have to position them at the same rank.

A more intricate issue is what to make of ordinals such as $\sqcup(3, 3, 3, \ldots)$. In spirit of the intuition given above it represents 3, that being the non-strict least upper

bound of $3, 3, 3, \ldots$.[1] We might like to exclude such representations and require that $\sqcup_i \alpha_i$ always represents a limit ordinal. This can be done by imposing a strict monotonicity condition on the limit sequences. Some order relation on tree ordinals is needed for that.

The following construction (Definitions 2 through 4) is due to Hancock (2008). In preparation for an extensional order relation on $\mathcal{TO}$, we define a structural strict order relation.

Definition 2. The set-valued function Φ defines the *predecessor indices* $\Phi(\alpha)$ *for* α by recursion on α:

$$\begin{aligned} \Phi(0) &= \varnothing \\ \Phi(\alpha^+) &= 1 + \Phi(\alpha) \\ \Phi(\sqcup_i \alpha_i) &= (\Sigma n \in \mathbb{N})\ \Phi(\alpha_n) \end{aligned} \qquad *$$

By $1 + A$ we mean the disjoint sum of the unit type 1 and A, where we use left and right a (for $a \in A$) as constructors of $1 + A$. The Σ-type $(\Sigma a \in A)B_a$ consists of the pairs $\langle a, b\rangle$ such that $a \in A$ and $b \in B_a$. Note that the set $\Phi(0)$ of predecessor indices for 0 has no inhabitants.

The predecessor indices of an ordinal α are essentially the paths on its tree structure starting from the root that cross at least one $^+$ constructor.

Definition 3. The function $\_[\_] : (\prod \alpha : \mathcal{TO})\ \Phi(\alpha) \to \mathcal{TO}$ defines the *predecessor* $\alpha[\iota]$ *of* α *indexed by* ι recursively on α:

$$\begin{aligned} \alpha^+[\text{left}] &= \alpha \\ \alpha^+[\text{right}\ \iota] &= \alpha[\iota] \\ \sqcup_i \alpha_i[\langle n, \iota\rangle] &= \alpha_n[\iota] \end{aligned} \qquad *$$

This structural predecessor function can be seen as defining a 'subtree' partial order on $\mathcal{TO}$. With it we are ready to define an extensional non-strict order relation on $\mathcal{TO}$ that classifies ordinals by rank.

Definition 4. We define the *order* $\preceq$ as a binary relation on $\mathcal{TO}$ by induction:

[1] Or, if we take the *strict* least upper bound, $\sqcup(3, 3, 3, \ldots)$ represents 4.

i. $0 \preceq \beta$ for every ordinal $\beta \in \mathcal{TO}$.
ii. For all $\alpha, \beta \in \mathcal{TO}$ and $\iota \in \Phi(\beta)$, if $\alpha \preceq \beta[\iota]$ then $\alpha^+ \preceq \beta$.
iii. For all $\alpha_0, \alpha_1, \alpha_2, \ldots, \beta \in \mathcal{TO}$, if $\alpha_n \preceq \beta$ for all $n \in \mathbb{N}$, then $\sqcup_i \alpha_i \preceq \beta$. ∗

An extensional equality on $\mathcal{TO}$ can now be defined on top of this order.

Definition 5. $\alpha, \beta \in \mathcal{TO}$ are *equal*, written $\alpha \simeq \beta$, if $\alpha \preceq \beta$ and $\beta \preceq \alpha$. ∗

The following proposition states that this equality implements our requirement of identifying $\sqcup_i i + 3$ and $\sqcup_i i \times 2$.

Proposition 1. $\sqcup_i i + 3 \simeq \sqcup_i i \times 2$.

Proof. Trivial. □

We extend the non-strict order on $\mathcal{TO}$ to a strict order.

Definition 6. $\alpha, \beta \in \mathcal{TO}$ are in *strict order*, written $\alpha \prec \beta$, if $\alpha \preceq \beta[\iota]$ for some $\iota \in \Phi(\beta)$. ∗

Now that the tree ordinals are equiped with a strict order, we can express the aforementioned monotonicity condition on the arguments of $\sqcup$ constructors.

Definition 7. We say that an ordinal is *well-formed* if it satisfies the *wf* property, defined by induction:

i. $wf\ 0$.
ii. For every $\alpha \in \mathcal{TO}$, if $wf\,\alpha$ then $wf\,\alpha^+$.
iii. For all $\alpha_0, \alpha_1, \alpha_2, \ldots \in \mathcal{TO}$, if for all $n, m \in \mathbb{N}$ we have $wf\,\alpha_n$ and that $n < m$ implies $\alpha_n \prec \alpha_m$, then $wf\ \sqcup_i \alpha_i$. ∗

The tree ordinal $\sqcup_i(3, 3, 3, \ldots)$ is, obviously, not well-formed.

2.2 Term Rewriting

We give a short introduction to the basic notions of infinitary term rewriting as required in the following chapters. For a more in-depth treatment of the theory of term rewriting, consult Terese (2003). Discussion of infinitary rewriting

specifically can be found in Terese (2003, Chapter 12) and Klop and de Vrijer (2005). In this section, we try to conform to definitions and notations from Terese (2003), but sometimes choose to follow more closely our Coq formalisation discussed in Chapter 3 when the two diverge.

Note that we routinely denote infinite terms by just 'terms' and infinitary rewriting by just 'rewriting'. We understand a 'sequence' to mean a finite or infinite list of objects and always explicitely write 'rewrite sequence' if this is what we mean to say.

2.2.1 Definition of a TRS

Definition 8. A *signature* Σ is a non-empty set of *function symbols* $f, g, \ldots$. Each function symbol f has a fixed natural number $\sharp f$, which we call its *arity*. A function symbol with arity 0 is also called a *constant*. $*$

Definition 9. The set of (possibly infinite) *terms* $Ter^{\infty}_{\Sigma}(X)$ over a signature Σ and a set of variables $\mathcal{X} = \{x, y, \ldots\}$ is defined by coinduction:

i. $x \in Ter^{\infty}_{\Sigma}(\mathcal{X})$ for every variable $x \in \mathcal{X}$.
ii. For every $f \in \Sigma$, if $t_1, \ldots, t_{\sharp f} \in Ter^{\infty}_{\Sigma}(\mathcal{X})$, then $f(t_1, \ldots, t_{\sharp f}) \in Ter^{\infty}_{\Sigma}(\mathcal{X})$. $*$

The symbol f is called the *root* of $f(t_1, \ldots, t_n)$ and the terms t_i are called the *arguments* of f. By $Var(t)$ we denote the set of variables occurring in t, and t is *closed* if $Var(t) = \varnothing$. If no variable occurs more than once in t, we say t is *linear*. Often, the set of variables $\mathcal{X}$ is left implicit and $Ter^{\infty}_{\Sigma}(\mathcal{X})$ is denoted simply by Ter^{∞}_{Σ}. By the set of *finite terms* Ter_{Σ} we mean the subset of well-founded terms of Ter^{∞}_{Σ}.

Preparing for the mechanised setting of Section 3 with its constrains of finite memory and computing time, we want to be precise about the notions of equality on infinite objects we employ. We consider terms to be equal if they are (i) bisimilar or (ii) pointwise equal up to every depth. According to Proposition 2 it does not matter which equality we use.

Definition 10. We define the *bisimilarity relation* $\leftrightarrows$ on Ter^{∞}_{Σ} by coinduction:

i. $x \leftrightarrows x$ for every variable $x \in \mathcal{X}$.
ii. For every $f \in \Sigma$, if $s_1 \leftrightarrows t_1, \ldots, s_{\sharp f} \leftrightarrows t_{\sharp f}$, then $f(s_1, \ldots s_{\sharp f}) \leftrightarrows f(t_1, \ldots, t_{\sharp f})$.

We say that s and t are *bisimilar* if $s \leftrightarrows t$. ∗

Definition 11. *(Pointwise) equality* of terms s and t *up to depth* d, written $s \equiv_{\leq d} t$, is defined by induction:

i. $s \equiv_{\leq 0} t$ for every $s, t \in Ter^{\infty}_{\Sigma}$.
ii. $x \equiv_{\leq d} x$ for every $d \in \mathbb{N}$ and $x \in \mathcal{X}$.
iii. For every $f \in \Sigma$, if $s_1 \equiv_{\leq d} t_1, \ldots, s_{\sharp f} \equiv_{\leq d} t_{\sharp f}$, then $f(s_1, \ldots s_{\sharp f}) \equiv_{\leq d+1} f(t_1, \ldots, t_{\sharp f})$.

Terms s, t are *pointwise equal*, written $s \equiv t$, if $s \equiv_{\leq d} t$ for every depth d. ∗

Proposition 2. $s \leftrightarrows t \;\Leftrightarrow\; s \equiv t$.

Proof. By induction on the depth of pointwise equality ($\Rightarrow$) and by coinduction on $\leftrightarrows$ ($\Leftarrow$). □

Definition 12. A *rewrite rule* ρ over a signature Σ is a pair $\langle l, r \rangle$ of finite terms in Ter_{Σ} (written $\rho : l \to r$). We only consider rewrite rules where l is not a variable and $Var(r) \subseteq Var(l)$. ∗

The two restrictions on rewrite rules are standard and prevent our theory from misbehaving in some particular ways. We say a rewrite rule is *left-linear* if its left-hand side is linear.

Definition 13. A *term rewriting system* (TRS) $\mathcal{R}$ is a pair $\langle \Sigma, R \rangle$ of a signature Σ and a finite set of rewrite rules R over Σ. ∗

2.2.2 Rewriting

Positions are sequences of natural numbers. The empty sequence is denoted by ϵ and ip is the prefixing of a sequence p with the number i.

Definition 14. The set of *positions* $\mathcal{P}os(t)$ *of a term* t is inductively defined by:

i. $\epsilon \in \mathcal{P}os(t)$ for every $t \in Ter^{\infty}_{\Sigma}$.
ii. For every $f \in \Sigma$ and $1 \leq i \leq \sharp f$, if $p \in \mathcal{P}os(t_i)$ then $ip \in \mathcal{P}os(f(t_1, \ldots, t_{\sharp f}))$.

The *subterm* of term t at position p, written $t|_p$, is inductively defined by (i) $t|_{\epsilon} = t$ and (ii) $f(t_1, \ldots, t_n)|_{ip} = t_i|_p$. Similarly, *updating* a term t at position p with term s, written $t[s]_p$, is defined by replacing the subterm $t|_p$ at position p in t with s. ∗

In contrast to Terese (2003), we do not define contexts as terms over an extended signature. Instead, a direct inductive definition of one-hole contexts is given, in line with our Coq development. See also Subsection 5.3.4.

Definition 15. The set of (one-hole) *contexts* Ctx^∞_Σ over a signature Σ is defined by induction:

i. $\Box \in Ctx^\infty_\Sigma$.
ii. For every $f \in \Sigma$ and $1 \leq i \leq \sharp f$, if $t_1, \ldots, t_{i-1}, t_{i+1}, \ldots, t_{\sharp f} \in Ter^\infty_\Sigma$ and $C \in Ctx^\infty_\Sigma$, then $f(t_1, \ldots, t_{i-1}, C, t_{i+1}, \ldots, t_{\sharp f}) \in Ctx^\infty_\Sigma$. *

Thus every context C has exactly one occurrence of the symbol $\Box$, called its *hole*. By the term $C[t]$ we mean the result of replacing the hole of C by t. We allow a slight abuse of notation by writing $t[\Box]_p$ for the context C with $C[t|_p] \equiv t$. We also assume obvious extensions to contexts for notions on terms (e.g. bisimilarity and $Var(C)$ and $\mathcal{P}os(C)$ for $C \in Ctx^\infty_\Sigma$). The *depth* of a context C is defined by the length of the (unique) position p with $C|_p = \Box$.

Definition 16. Given a signature Σ and a set of variables $\mathcal{X}$, a *substitution* σ is a mapping from $\mathcal{X}$ to $Ter^\infty_\Sigma(\mathcal{X})$. It can be generalised to a mapping $\bar{\sigma} : Ter^\infty_\Sigma(\mathcal{X}) \to Ter^\infty_\Sigma(\mathcal{X})$ corecursively:

$$\bar{\sigma}(x) = \sigma(x)$$
$$\bar{\sigma}(f(t_1, \ldots, t_n)) = f(\bar{\sigma}(t_1), \ldots, \bar{\sigma}(t_n)) \quad *$$

Since $\bar{\sigma}$ is completely defined by σ we refer to both as 'the' substitution σ. Applying a substitution σ to a term t is usually written t^σ and the result is called an *instance* of t.

If we view a rewriting rule $\rho : l \to r$ as a *scheme*, an *instance* of ρ can be obtained by applying a substitution σ. The result is the *atomic* rewrite step $l^\sigma \to_\rho r^\sigma$. We call l^σ a (ρ-) *redex* and r^σ its *contractum*. An atomic rewrite step can be placed in a context, forming a rewrite step.

Definition 17. A *rewrite step* $C[l^\sigma] \to_\rho C[r^\sigma]$ according to the rewrite rule ρ consists of rewriting the redex obtained from ρ and substitution σ to its contractum in a context C. *

The *depth* of a rewrite step is the depth of its context. We call $\to_\rho$ the *one-step rewriting relation* generated by ρ. The one-step rewriting relation $\to$ of a TRS

$\mathcal{R}$ with rewrite rules R is defined as the union of $\{\to_\rho \mid \rho \in R\}$.

Related to equality of (infinite) terms is equality of rewrite steps. Since rewrite steps contain possibly infinite contexts, it makes sense to define equality of rewrite steps via equality of their contexts. This is made precise in the following definition and the resulting equality is used in Subsection 3.3.1.

Definition 18. Two rewrite steps are defined to be *equal* if

i. they use the same rewrite rule ρ,
ii. their contexts are bisimilar, and
iii. their substitutions agree on all variables in ρ. *

Definition 19. A *rewrite sequence* of ordinal length α is a sequence of rewrite steps $(t_\beta \to t_{\beta^+})_{\beta<\alpha}$. *

This definition only makes sense if we somehow require that for every limit ordinal $\lambda \prec \alpha$, the terms $(t_\beta)_{\beta<\lambda}$ approach t_λ in the limit and usually an even further restrictions is desirable. We define the notion of Cauchy convergence and, using that, four conditions on rewrite sequences.

Definition 20. Let λ be a limit ordinal. A sequence of terms $(t_\beta)_{\beta<\lambda}$ *(Cauchy-) converges to the term* t if for every depth d there exists $\alpha \prec \lambda$ such that for all $\alpha \preceq \beta \prec \lambda$ we have $t_\beta \equiv_{\le d} t$. *

Definition 21. A rewrite sequence $(t_\beta \to t_{\beta^+})_{\beta<\alpha}$ of length α is

i. *weakly continuous* if for every limit ordinal $\lambda \prec \alpha$, the sequence $(t_\beta)_{\beta<\lambda}$ converges to the term t_λ,
ii. *strongly continuous* if it is weakly continuous and for every limit ordinal $\lambda \prec \alpha$, the depth of the rewrite steps $(t_\beta \to t_{\beta^+})_{\beta<\lambda}$ tends to infinity,
iii. *weakly convergent* if for every limit ordinal $\lambda \preceq \alpha$, the sequence $(t_\beta)_{\beta<\lambda}$ converges to the term t_λ, and
iv. *strongly convergent* if it is weakly convergent and for every limit ordinal $\lambda \preceq \alpha$, the depth of the rewrite steps $(t_\beta \to t_{\beta^+})_{\beta<\lambda}$ tends to infinity. *

We write $t_0 \twoheadrightarrow t_\alpha$ if there exists a strongly convergent rewrite sequence $(t_\beta \to t_{\beta^+})_{\beta<\alpha}$ or $t_0 \to^\alpha t_\alpha$ if we want to stress its length. The *convertibility* relation is defined as the equivalence closure of $\twoheadrightarrow$.

2.2.3 Normal Forms and Orthogonality

Definition 22. Let $\mathcal{R}$ be a TRS.

i. A term t is a *normal form* if there is no rewrite step from t. We say t is a normal form *of* s if $s \twoheadrightarrow t$ and t is a normal form.
ii. $\mathcal{R}$ has the property of *unique normal forms* (UN^{∞}) if $t \equiv u$ for every two convertible normal forms t and u.
iii. $\mathcal{R}$ has the property of *unique normal forms with respect to rewriting* ($\mathrm{UN}^{\twoheadrightarrow}$) if for all terms s, we have $t \equiv u$ for all normal forms t and u of s. ∗

Obviously we have that UN^{∞} implies $\mathrm{UN}^{\twoheadrightarrow}$. One source of non-unique normal forms is the interference of two redex occurrences in a term. Contracting one of them may result in a term where (a descendant of) the other redex is no longer present, possibly losing confluence. This phenomenon is made precise in the following definition.

Definition 23. We say two rewrite rules $\rho_1 : l_1 \to r_1$ and $\rho_2 : l_2 \to r_2$ have *overlap* if there exists a non-variable position $p \in \mathcal{P}os(l_1)$ such that $l_1|_p$ and l_2 have a common instance. (We exclude the trivial case of overlap between a rewrite rule and itself at the root position.) Let σ, τ be substitutions such that ${l_1}|_p^{\sigma} \equiv {l_2}^{\tau}$ is a most general common instance of $l_1|_p$ and l_2, and without loss of generality assume that $dom(\sigma) = Var(l_1|_p)$ and $Var({l_1}|_p^{\sigma}) \cap Var(l_1[\Box]_p) = \varnothing$. Then $\langle {l_1}^{\sigma}[{r_2}^{\tau}]_p, {r_1}^{\sigma} \rangle$ is called a *critical pair of* ρ_1 *with* ρ_2. A critical pair $\langle s, t \rangle$ is called *trivial* if $s \equiv t$. ∗

Critical pairs are unique up to renaming of variables. Using these notions we can define some useful classes of term rewriting systems.

Definition 24. A TRS is called

i. *left-linear* if all its rewrite rules are left-linear,
ii. *orthogonal* if it is left-linear and there are no critical pairs, and
iii. *weakly orthogonal* if it is left-linear and all critical pairs are trivial. ∗

A fundamental result in the theory of infinitary rewriting is the Compression Lemma.

LEMMA 1 (Compression). Every strongly convergent rewrite sequence in a left-linear TRS can be compressed to length less than or equal to ω.

PROOF. By transfinite induction on the length of the rewrite sequence. See for example Terese (2003, Theorem 12.7.1, page 689) or Endrullis et al. (2010). □

Orthogonal rewrite systems enjoy the UN$^\infty$ property (Kennaway, Klop, Sleep, and de Vries, 1995; Klop and de Vrijer, 2005). In Chapter 4 we formalise the counterexample to UN$^\infty$ for weakly orthogonal TRSs from Endrullis et al. (2010).

Chapter 3

A Mechanical Formalisation

In this chapter we present a formalisation of some of the notions from Chapter 2 in the Coq proof assistant. Section 3.1 translates the tree ordinals from Subsection 2.1.1 to Coq. Coinductive terms are defined in Section 3.2. In Section 3.3 we present a novel representation of transfinite rewrite sequences based on the structure of the tree ordinals. This we regard as the main contribution of this thesis.

A short introduction to Coq is included in Appendix A. In the Coq code fragments, we take some notational liberties in favour of readability. Sometimes we omit (part of) the type information. We also freely use infix notations without declaration. Furthermore, variable and definition names are typeset liberally.

Some definitions have implicit arguments, meaning those arguments can be inferred by the system from the context. As an example, consider the inductive type nat^+ whose constructor takes as arguments a natural number n and a proof that n is greater than 0.

```
Inductive nat⁺ : Set :=
  | Pos : ∀ n : nat, 0 < n → nat⁺.
```

The argument n of Pos can be implicit, since it can be inferred from (the type of) the other argument. If H has type $0 < 3$, we can write Pos H instead of Pos 3 H.

Related work are the CoLoR (Blanqui and Koprowski, 2010) and Coccinelle (Contejean, Courtieu, Forest, Pons, and Urbain, 2007) projects, libraries on

finitary rewriting and termination, and the representation of ordinal numbers up to ϵ_0 and Γ_0 in Cantor and Veblen normal form by Castéran (2006).

3.1 Ordinal Numbers

In the theory of infinitary rewriting, the lengths of rewrite sequences play a central role. One might even suspect that any representation of transfinite rewrite sequences needs a representation of ordinal numbers. But this is not the case.

Consider as an illustration the example of finite lists. They can be naturally represented inductively, without the need for a representation of natural numbers. The usual inductive definition of lists, using constructors Nil and Cons, can be seen as a generalisation of the natural numbers, defined inductively using constructors Zero and Successor. The generalisation consists of labeling the Cons constructors with list members.

Likewise, we now turn to the definition of tree ordinals as a case study in preparation for the definition of transfinite rewrite sequences in Section 3.3.

We define the ordinal numbers using the representation of tree ordinals (cf. Definition 1) in Coq by ord.

```
Inductive ord : Set :=
  | Zero : ord
  | Succ : ord → ord
  | Limit : (nat → ord) → ord.
```

Arithmetic operations on ordinals, such as addition, are easily defined.

```
Fixpoint + (α β : ord) : ord :=
  match β with
  | Zero ⇒ α
  | Succ β ⇒ Succ (α + β)
  | Limit f ⇒ Limit (fun n ⇒ α + (f n))
  end.
```

In fact, all definitions from Subsection 2.1.1 translate directly to Coq code. We can now prove basic properties of $\preceq$, for example that it is transitive and that, for the finite ordinals, it coincides with the standard order on the natural numbers.[1]

[1] Although n and m have type nat and $\preceq$ has type ord → ord → Prop, we can state the lemma in this concise way by defining the trivial coercion from nat to ord.

```
Lemma ≤trans : ∀ α β γ, α ≤ β → β ≤ γ → α ≤ γ.
Lemma ≤nat : ∀ n m, n ≤ m ↔ n ≤ m.
```

Recalling our discussion in Subsection 2.1.1 of limit ordinals whose sequences do not actually approximate to a limit ordinal, we consider the lemma ≤zero_right as an example of this issue.

```
Lemma ≤zero_right : ∀ α β, α ≤ Zero → α ≤ β.
```

We would like to strengthen this, but cannot, since nothing denies α from being the tree ordinal $\sqcup(0, 0, 0, \ldots)$ (which has the same rank as 0). We therefore turn to a subset of the tree ordinals where we restrict limit sequences to be strictly monotonic. This restriction is encoded in the wf (well-formedness) property. The Σ-type ord^wf defines the resulting subset.

```
Fixpoint wf α : Prop :=
  match α with
  | Zero ⇒ True
  | Succ β ⇒ wf β
  | Limit f ⇒ ∀ n, wf (f n) ∧ ∀ m, n < m → f n ≺ f m
  end.
Definition ord^wf : Set := { α : ord | wf α }.
```

Now we can prove the stronger result we were looking for.[1]

```
Lemma ≤^wf_zero_right : ∀ α : ord^wf, α ≤ Zero → α = Zero.
```

3.2 Coinductive Terms

We define the type term of infinite terms with function symbols in F and variables in X.

```
CoInductive term : Type :=
  | Var : X → term
  | Fun : ∀ f : F, vector term (arity f) → term.
```

The objects of a coinductive type can only be built in some restricted way to ensure productivity of their construction. This restriction implies a technical difficulty in the definition of the vector type, which is discussed in Section A.4.

[1] Again, defining a simple coercion from ord^wf to ord (first Σ-type projection) lets us state this lemma concisely.

For now, we assume vector to implement dependently typed lists (the type depending on their length).

The standard equality defined in Coq, equivalent to Leibniz' equality and written =, does not suffice for establishing that two terms are equal, given that the only way to build infinite objects is by corecursion. Because the amount of memory available is finite, we can only unfold the corecursive definition finitely many times, and then still be left with a non-normal form. Simply comparing such definitions will not do, since the corecursive construction of any given infinite object is not unique. To this end, we define two extensional equalities on term, following Definitions 10 and 11. The coinductive relation $\leftrightarrows$ defines bisimilarity and pointwise equality is defined by $\equiv$ inductively.

```
CoInductive ⇆ : term → term → Prop :=
  | ⇆Var : ∀ x, Var x ⇆ Var x
  | ⇆Fun : ∀ f v w, (∀ i, v i ⇆ w i) → Fun f v ⇆ Fun f w.
```

Any proof of two infinite terms being bisimilar is an infinite proof, in the sense that the proof term is built by corecursion.

```
Inductive ≡≤ : nat → term → term → Prop :=
  | teut0 : ∀ s t, s ≡≤0 t
  | teutVar : ∀ d x, Var x ≡≤d Var x
  | teutFun : ∀ d f v w, (∀ i, v i ≡≤d w i) → Fun f v ≡≤S d Fun f w.
Definition s ≡ t := ∀ d, s ≡≤d t.
```

We can prove that $\leftrightarrows$ and $\equiv$ are the same relation, and that indeed it is an equivalence.

```
Lemma term_bis_term_eq : ∀ s t, s ⇆ t ↔ s ≡ t.
Lemma ⇆refl : ∀ t, t ⇆ t.
Lemma ⇆symm : ∀ s t, s ⇆ t → t ⇆ s.
Lemma ⇆trans : ∀ s t u, s ⇆ t → t ⇆ u → s ⇆ u.
```

In Section 3.3 we need some notion of convergence for functions of type nat → term. We implement Definition 20 in Coq for sequences of length ω.

```
Definition converges (f : nat → term) (t : term) : Prop :=
  ∀ d, ∃ n, ∀ m, n ≤ m → f m ≡≤d t.
```

The definitions of finite term, rewrite rule, TRS, and left-linearity from Subsection 2.2.1 translate to Coq directly. We define lhs and rhs to be first and second projection on rewrite rules, respectively.

The type of contexts is inductively defined, where the hole always occurs at a finite depth.

```
Inductive context : Type :=
  | □ : context
  | CFun : ∀ (f : F) (i j : nat), i + S j = arity f →
             vector term i → context → vector term j → context.
```

Applying a substitution σ to a term t is defined by corecursion over t. We also use the notation t^{σ} for substitute σ t.

```
Definition substitution := X → term.

CoFixpoint substitute (σ : substitution) (t : term) : term :=
  match t with
  | Var x ⇒ σ x
  | Fun f args ⇒ Fun f (vmap (substitute σ) args)
  end.
```

We apply the recursive function fill (not shown here) to a context C and a term t (written $C[t]$) to replace the hole in C with t.

Positions are represented by simple lists of natural numbers. This means the subterm at some position in some term may not actually exist. For this reason we employ option types in functions that do a lookup by position (functions in Coq are always *total*). For further discussion of positions, see Section 5.3.

```
Fixpoint dig (t : term) (p : position) {struct p} : option context :=
  match p with
  | nil ⇒ Some □
  | n :: p ⇒ match t with
      | Var _ ⇒ None
      | Fun f args ⇒ match lt_ge_dec n (arity f) with
          | left h ⇒ match dig (vnth h args) p with
              | None ⇒ None
              | Some C ⇒ Some (CFun f (lt_plus_minus_r h)
                                     (vtake (lt_le_weak n (arity f) h) args)
                                     C
                                     (vdrop h args))
              end
          | right _ ⇒ None
          end
      end
  end.
```

```
Fixpoint subterm (t : term) (p : position) {struct p} : option term :=
  match p with
  | nil ⇒ Some t
  | n :: p ⇒ match t with
                | Var _ ⇒ None
                | Fun f args ⇒ match lt_ge_dec n (arity f) with
                                 | left h ⇒ subterm (vnth h args) p
                                 | right _ ⇒ None
                                 end
                end
  end.
```

Now subterm $t\ p$ gives the subterm of t (if it exists) and dig $t\ p$ gives the context C (if it exists) that is t with subterm $t\ p$ replaced by $\Box$ at position p.

3.3 Transfinite Rewrite Sequences

In this section we present the essence of our development. Rewrite sequences of length α are represented using the tree structure of the tree ordinal α. Much of the definitions on ordinals are lifted to rewrite sequences and we once more come to a notion of well-formedness. The resulting representation is discussed in relation to the traditional theory of rewriting in Section 5.2.

Throughout this section, we let $\mathcal{R}$ be a fixed TRS. We define the type of steps using rewrite rules in $\mathcal{R}$, parameterised by their source and target terms. Some flexibility in the form of bisimilarity is allowed, motivated in Subsection 5.3.3.

```
Inductive →ℛ : term → term → Type :=
  | Step : ∀ (s t : term) (ρ : rule) (C : context) (σ : substitution),
             ρ ∈ ℛ →
             C[(lhs ρ)^σ] ↔ s →
             C[(rhs ρ)^σ] ↔ t → (s →ℛ t).
```

For the translation of Definition 18 (equality of steps) to Coq, we assume the lifting of bisimilarity to contexts and that substitution_eq defines agreement of substitutions on a given list of variables.

```
Definition ≈ (s t : term) (π : s →ℛ t) (u v : term) (o : u →ℛ v) : Prop :=
  match π, o with
  | Step _ _ ρ C σ _ _ _, Step _ _ ρ' C' σ' _ _ _ ⇒
    C ↔ C' ∧ ρ = ρ' ∧ substitution_eq (vars (lhs ρ)) σ σ'
  end.
```

We describe a way to define rewrite sequences as an inductive type. A rewrite sequence of length α can be represented by the tree ordinal α where we label every occurrence of the $^+$ constructor with a rewrite step. To ensure that successive steps have the same target and source terms, respectively, we include the source and target terms of the rewrite sequence in its type and label accordingly.

At this point, it is not immediately clear what the type of the limit constructor should be. Following the tree ordinals, we think of a rewrite sequence as a countably branching tree with every branching node representing the least upper bound of its branches. As a first step towards a type of rewrite sequences, we write down an incomplete try. Note that the Cons constructor appends (not prepends) a step to a sequence.

```
Inductive →↠ℛ : term → term → Type :=
  | Nil : ∀ t, t →↠ℛ t
  | Cons: ∀ s t u, (s →↠ℛ t) → (t →ℛ u) → (s →↠ℛ u)
  | Lim : ∀ s t, (nat → s →↠ℛ ?) → (s →↠ℛ t).
```

This is not yet satisfying, because we cannot fix a value for **?**. We complete the type for Lim as follows. First, we parameterise it with the target terms of the branches. Second, we add the condition that these terms must converge to the target term t.

```
  | Lim : ∀ s t (ts : nat → term),
          (∀ n, s →↠ℛ ts n) → converges ts t → (s →↠ℛ t)
```

Of course, the branches of a Lim constructor may still not actually approximate to a rewrite sequence (of length a limit ordinal). The intuition is that each branch should extend on its preceding ones. This would correspond to the wf property we defined on ord, where we lift $\prec$ to a strict prefix relation on $\twoheadrightarrow_{\mathcal{R}}$. We return to this issue in Subsection 3.3.2, but first consider the definition of an embedding relation on $\twoheadrightarrow_{\mathcal{R}}$.

3.3.1 Embeddings of Rewrite Sequences

We lift the notions of predecessor and predecessor indices to the domain of transfinite rewrite sequences. The set of predecessor indices is easily defined as pred_type.

```
Fixpoint pred_type s t (φ : s →↠ℛ t) : Type :=
  match φ with
  | Nil _ ⇒ empty
  | Cons _ _ _ ψ _ ⇒ unit + pred_type ψ
  | Lim _ _ _ f _ ⇒ { n : nat & pred_type (f n) }
  end.
```

The predecessor indices defined by pred_type point to a specific occurrence of the Cons constructor in a rewrite sequence. This constructor does not only contain a rewrite sequence (analogous to an ordinal in the ord case), but also a rewrite step. The pred function gives us both the rewrite sequence and the step pointed to by a predecessor index. For the type checker to accept the definition, we use a Σ-type that contains this pair, parameterised by the source and target terms of the rewrite step.

```
Fixpoint pred s t (φ : s →↠ℛ t) (ι : pred_type φ) :
    { ts : term × term & (s →↠ℛ fst ts) × (fst ts →ℛ snd ts) } :=
  match φ with
  | Nil _ ⇒ (empty_rect _) ι
  | Cons _ u t ψ π ⇒ match ι with
                     | inl tt ⇒ existT _ (u, t) (ψ, π)
                     | inr κ ⇒ pred ψ κ
                     end
  | Lim _ _ _ f _ ⇒ match ι with
                    | existT n κ ⇒ pred (f n) κ
                    end
  end.
```

In an effort to prevent getting lost in a syntactical labyrinth, we define the following notational shortcuts:

SHORT	EXPLANATION	DEFINITION
$\varphi[\iota]$	location in φ indexed by ι	pred φ ι
$\varphi[\iota]^{\text{SEQ}}$	rewrite sequence in φ indexed by ι	fst (projT2 (pred φ ι))
$\varphi[\iota]^{\text{STP}}$	step of φ indexed by ι	snd (projT2 (pred φ ι))
$\varphi[\iota]^{\text{L}}$	source term of $\varphi[\iota]^{\text{STP}}$	fst (projT1 (pred φ ι))
$\varphi[\iota]^{\text{R}}$	target term of $\varphi[\iota]^{\text{STP}}$	snd (projT1 (pred φ ι))

As an example of predecessor indexing, consider the graphical representation of a rewrite sequence φ of length $\omega+2$ and its predecessor index ι = inr (inr ⟨4, inl⟩) in Figure 3.1. The initial part of length ω is represented by a series of finite

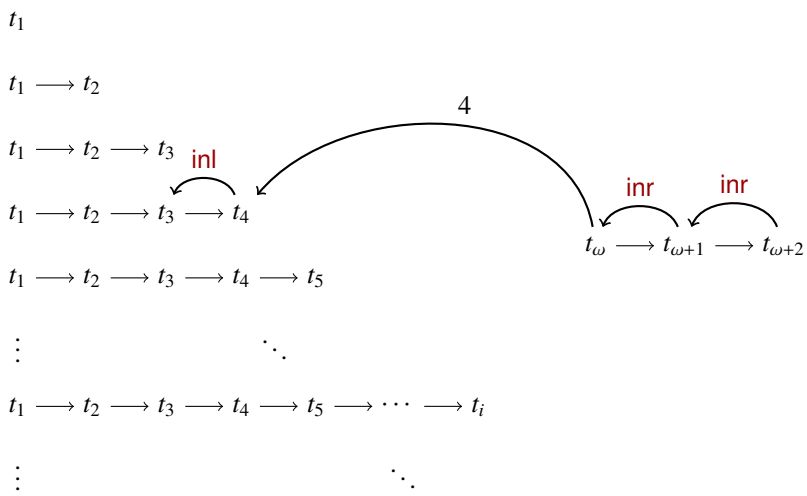


FIGURE 3.1. Example of a rewrite sequence and predecessor index.

rewrite sequences, each one extending on the previous one by one step. The sequence of terms $\{t_1, t_2, t_3, \ldots\}$ converges to the term t_ω. Here, $\varphi[\iota]^{\text{SEQ}}$ is a rewrite sequence from t_1 to t_3 and $\varphi[\iota]^{\text{STP}}$ is a step from t_3 to t_4.

Having a closer look at the order $\preceq$ on the tree ordinals, we can see that it really defines embeddings of their tree structures. This is due to clause II of Definition 4. In this clause, two occurrences of the $^+$ constructor (one in both ordinals) are cancelled out against each other, but the positions of these occurrences in their respective ordinals do not necessarily correspond. Since occurrences of $^+$ carry no additional information, this has no effect on the resulting relation.

What this means for a translation of $\preceq$ to the domain of our inductively defined rewrite sequences is that, indeed, we get an embedding relation. We only have to make sure that in the Cons case, we cancel out two equal steps against each other. We say that φ is embedded in ψ (written $\varphi \sqsubseteq \psi$) if ψ can be obtained from φ by inserting any number of steps in φ. We distinguish between inserting a step (i) before the first step, (ii) after the last step, and (iii) in between steps in a rewrite sequence. Note that any steps inserted consecutively in between steps necessarily form a cycle, because of the typing constraints in the definition of rewrite sequence.

```
Inductive ⊑ : ∀ s t u v, (s →ℛ t) → (u →ℛ v) → Prop :=
  | ⊑Nil : ∀ s u v (ψ : u →ℛ v),
                    Nil s ⊑ ψ
  | ⊑Cons : ∀ s t u v (ψ : u →ℛ v) (ι : pred_type ψ) (φ : s →ℛ ψ[ι]ᴸ)
            (π : ψ[ι]ᴸ →ℛ t),
                    φ ⊑ ψ[ι]ˢᴱᵠ →
                    π ≈ ψ[ι]ˢᵀᴾ →
                    Cons φ π ⊑ ψ
  | ⊑Lim : ∀ s t u v (ts : nat → term) (f : ∀ n, s →ℛ ts n)
            (c : converges ts t) (ψ : u →ℛ v),
                    (∀ n, f n ⊑ ψ) →
                    Lim f c ⊑ ψ.
```

Analogous to the strict order $\prec$ on ordinals, we define a strict embedding relation $\sqsubset$ on rewrite sequences.

```
Definition ⊏ s t u v (φ : s →ℛ t, ψ : u →ℛ v) := ∃ ι, φ ⊑ ψ[ι]ˢᴱᵠ.
```

Note that, while non-strictly embedded rewrite sequences may differ in any of the three ways defined above, strictly embedded rewrite sequences always differ in their last step. Thus, if φ is strictly embedded in ψ then ψ can be obtained from φ by inserting any number of steps in φ, but at least one after the last step.

3.3.2 Well-formed Rewrite Sequences

The wf property on ord is defined in Section 3.1 to rule out a certain class of ordinal representations. This issue translates directly to our inductive representation of rewrite sequences. We define a well-formedness property wf on rewrite sequences, using the strict embedding relation $\sqsubset$.

```
Fixpoint wf s t (φ : s →ℛ t) : Prop :=
  match φ with
  | Nil _ ⇒ True
  | Cons _ _ ψ _ _ ⇒ wf ψ
  | Lim _ _ f _ _ ⇒ (∀ n, wf (f n)) ∧ ∀ n m, n < m → f n ⊏ f m
  end.
```

On page 23, we define the Lim constructor with the intuition that each of its branches should extend on the preceding ones. Naturally, we would implement this condition using a strict prefix relation on rewrite sequences, but the strict embedding relation $\sqsubset$ is also satisfying for this purpose.

Consider an instance of Lim, satisfying wf, with branches f. For every $n < m$, we have $f\ n \sqsubset f\ m$. Thus there is a predecessor sequence of $f\ m$ that can be obtained from $f\ n$ by inserting any number of steps in $f\ n$. Steps inserted before the last step of $f\ n$ must form a cycle (note that all branches start at the same term). Therefore, $f\ m$ can be obtained from $f\ n$ by adding one or more steps at the end of $f\ n$ and possibly inserting cycles at other positions of $f\ n$. This shows that, ignoring cycles, $\sqsubset$ actually defines a strict prefix relation on the branches of f.

There is still an important omission in our formalisation though: even rewrite sequences satisfying wf are not necessarily convergent. How the convergence conditions from Definition 21 relate to our formalisation is discussed in Section 5.2.

3.3.3 Combining Rewrite Sequences

With the Cons constructor, we can extend a rewrite sequence with one step at the end. Dually, snoc extends a rewrite sequence with one step at the start. It is the analogue of $1 + \alpha$ on ordinals.

snoc is recursive in its right argument, but for the Coq type checker to accept our definition, we must write it such that it consumes this argument first.[1] Hence, we use an auxiliary function snoc_rec.

```
Fixpoint snoc_rec s t u (φ : t →ₐ u) : (s →ₐ t) → (s →ₐ u) :=
  match φ with
  | Nil _ ⇒ fun π ⇒ Cons (Nil s) π
  | Cons _ _ ψ _ o ⇒ fun π ⇒ Cons (snoc_rec ψ π) o
  | Lim _ _ f u c ⇒ fun π ⇒ Lim (fun o ⇒ snoc_rec (f o) π) c
  end.

Definition snoc s t u (π : s →ₐ t) (φ : t →ₐ u) : s →ₐ u := snoc_rec φ π.
```

A related operation is concatenation of rewrite sequences, the analogue of addition on ordinals. It is defined in the same way as snoc.

[1] The reason for this is rather technical, but the idea is that the return type nicely follows the case analysis on the rewrite sequence in the `match` construction. We also give some hints to the Coq type checker that are not shown here.

```
Fixpoint concat_rec s t u (ψ : t →ₐ u) : (s →ₐ t) → (s →ₐ u) :=
  match ψ with
  | Nil _ ⇒ fun φ ⇒ φ
  | Cons _ _ ψ _ π ⇒ fun φ ⇒ Cons (concat_rec ψ φ) π
  | Lim _ _ f u c ⇒ fun φ ⇒ Lim (fun o ⇒ concat_rec (f o) φ) c
  end.

Definition concat s t u (φ : s →ₐ t) (ψ : t →ₐ u) : s →ₐ u :=
  concat_rec ψ φ.
```

Well-formedness is preserved under concatenation.

```
Lemma concat_wf : ∀ s t u (φ : s →ₐ t) (ψ : t →ₐ u),
                    wf φ → wf ψ → wf (concat φ ψ).
```

3.4 Properties of Terms and TRSs

We define some predicates on terms and TRSs. Again, we let $\mathcal{R}$ be a fixed TRS throughout this section.

We work with a somewhat relaxed definition of critical pairs. First, we do not require the common instance to be a most general one. Second, the substitution σ might not be minimal and might not introduce only fresh variables (cf. Definition 23). The effect of this relaxation is that for every critical pair, we have a series of critical pairs by this Coq definition. This is precise enough for our present purposes, however, since it has no effect on questions such as *are there critical pairs?* or *are all critical pairs trivial?*.

```
Definition critical_pair (R : trs) (t₁ t₂ : term) : Prop :=
  ∃ ρ₁ : rule, ∃ ρ₂ : rule, ∃ p : position, ∃ σ, ∃ τ,
      ρ₁ ∈ R ∧ ρ₂ ∈ R ∧ (ρ₁ = ρ₂ → p ≠ nil) ∧
      match subterm (lhs ρ₁) p, dig (lhs ρ₁)^σ p with
      | Some s, Some C ⇒ is_var s = false ∧ s^σ ↔ (lhs ρ₂)^τ ∧
                         t₁ ↔ C[(rhs ρ₂)^τ] ∧ t₂ ↔ (rhs ρ₁)^σ
      | _, _ ⇒ False
      end.
```

Now we can in a straightforward manner define the properties of orthogonality and weak orthogonality.

```
Definition orthogonal (R : trs) : Prop :=
  trs_left_linear R ∧ ∀ t₁ t₂, ¬ critical_pair t₁ t₂.
```

```
Definition weakly_orthogonal (ℛ : trs) : Prop :=
  trs_left_linear ℛ ∧ ∀ t1 t2, critical_pair t1 t2 → t1 ↔ t2.
```

Next we define when a term is a normal form and when we have unique normal forms.

```
Definition normal_form t : Prop :=
  ¬ ∃ C : context, ∃ ρ : rule, ∃ σ : substitution, ρ ∈ ℛ ∧ C[(lhs r)^σ] ↔ t.
Definition unique_normal_forms : Prop :=
  ∀ s t u (φ : s →ℛ t) (ψ : s →ℛ u),
    wf φ → wf ψ → normal_form t → normal_form u → t ↔ u.
```

Note that the unique_normal_forms definition is only a translation of the $UN^{\rightarrow}$ property, not of the more general UN^{∞} property (see also Definition 22).

Chapter 4

Unique Normal Forms in Weakly Orthogonal Systems

Finitary orthogonal term rewriting systems have unique normal forms. In fact, weak orthogonality is enough to establish this property for finitary systems (Terese, 2003, Chapter 4). To what extent can these results be lifted to infinitary rewriting?

In the infinitary setting, orthogonal TRSs exhibit the infinitary unique normal forms (UN^∞) property (Kennaway et al., 1995; Klop and de Vrijer, 2005). We might expect this property to generalise to weakly orthogonal systems. After all, the motivation for allowing trivial critical pairs in these systems is that, intuitively, they are witnesses of harmless overlap. However, this intuition turns out to be unjust for the infinitary case.

4.1 A Counterexample

We describe a simple counterexample showing that weak orthogonality does not imply the UN^∞ property (Endrullis et al., 2010).

We work in a signature with unary function symbols D and U.[1] In the notation

[1] We can think of D and U as 'down' and 'up'. The original formulation of this TRS uses P and S ('predecessor' and 'successor'), but to avoid notational conflicts with the S constructor for nat in Coq, we proceed with this modification.

of terms, we omit the brackets around arguments and assume right-associativity of function symbol application, e.g. writing DUx for $D(U(x))$. A notation for finite repetitions of a function symbol f terminated by a term t is defined by (i) $f^0 t = t$ and (ii) $f^{n+1} = f f^n t$. The infinite nesting $fff\ldots$ of f is written f^ω.

Consider the TRS consisting of the two left-linear rewrite rules ρ_1 and ρ_2:

$$\rho_1 : DUx \to x \qquad\qquad \rho_2 : UDx \to x$$

This system has two critical pairs $\langle Dx, Dx\rangle$ and $\langle Ux, Ux\rangle$, both of which are trivial, establishing weak orthogonality. The infinite term $\upsilon = D^1U^2D^3U^4\ldots$ has two normal forms. It rewrites to U^ω in ω many ρ_1-steps and to D^ω in ω many ρ_2-steps. This contradicts $\mathrm{UN}^{\twoheadrightarrow}$ and therefore also UN^∞.

Other interesting properties of this TRS (e.g. weak normalisation is not preserved under rewriting) and a translation to the infinitary $\lambda\beta\eta$-calculus are discussed by Endrullis et al. (2010).

4.1.1 Rewriting υ to U^ω

We show briefly what rewriting υ to U^ω amounts to. Rewriting υ to D^ω is done in a similar way. An obvious way to define υ by corecursion is via auxiliary terms υ'_n parameterised by n as follows:

$$\upsilon = \upsilon'_0 \qquad\qquad \upsilon'_n = U^n D^{n+1} \upsilon'_{n+2}$$

But a more useful definition for our present purposes, and the one we stick with, is the slight reformulation:

$$\upsilon = \upsilon'_0 \qquad\qquad \upsilon'_n = D^{2n+1} U^{2n+2} \upsilon'_{n+1}$$

For any term t and natural numbers n, m we have $U^n D^{m+1} U^{m+1} t \to_{\rho_1} U^n D^m U^m t$ and thus $U^n D^m U^m t \twoheadrightarrow U^n t$ by iterating m such steps. Instantiating m with $2n+1$ and t with $U\upsilon'_{n+1}$, we obtain $U^n \upsilon'_n \twoheadrightarrow U^{n+1} \upsilon'_{n+1}$ for any n. Concatenating these sequences, iterating n from 0 onwards, we arrive at $\upsilon \twoheadrightarrow U^\omega$.

4.2 The Counterexample in Coq

We implement the counterexample from Section 4.1 using the Coq development described in Chapter 3.

The rewrite rules ρ_1 and ρ_2 are straightforwardly defined and shown to be left-linear. By a simple proof we obtain that all critical pairs are trivial and hence that the TRS is weakly orthogonal.

```
Lemma wo_R : weakly_orthogonal R.
```

We introduce the notation f @ t to mean Fun f (vcons t (vnil term)). For brevity, in the following discussion we focus on the function symbol U and omit analoguous definitions using the function symbol D. The infinite term U^ω is defined by corecursion and finite repetitions $U^n t$ are defined by recursion (and are assumed to generalise to contexts with the same notation).

```
CoFixpoint U^ω : term := U @ U^ω.

Fixpoint U^n t :=
  match n with
  | O ⇒ t
  | S n ⇒ U @ (U^n t)
  end.
```

Unfortunately, υ is not as easily defined. Although clearly productive, direct translations of the corecursive definitions in Subsection 4.1.1 do not satisfy Coq's guardedness condition (see also Section A.4). The conclusion of a *trial and error* approach is that we must use anonymous cofix constructions. The definition we proceed with is the following:

```
CoFixpoint υ' n : term :=
  (cofix Ds (d : nat) :=
    match d with
    | O ⇒ D @ (cofix Us (u : nat) :=
                 match u with
                 | O ⇒ υ' (S n)
                 | S u ⇒ U @ U @ (Us u)
                 end) (S n)
    | S d ⇒ D @ D @ (Ds d)
    end) n.

Definition υ := υ' 0.
```

We now prove that U^ω and D^ω are (distinct) normal forms. This is a tedious proof that essentially consists of analysing the occurrence of a redex in these terms, yielding that there can not be such an occurrence.

```
Lemma nf_{U^ω} : normal_form (system := R) U^ω.
Lemma nf_{D^ω} : normal_form (system := R) D^ω.
Lemma neq_{D^ω}^{U^ω} : ¬ U^ω ↔ D^ω.
```

Constructing a rewrite sequence from υ to U^ω is done in much the same way as described in Subsection 4.1.1. First, we define the parameterised step that is used in the rewrite sequence. It eliminates one pair of D, U constructors in a term of the form $U^n D^{m+1} U^{m+1} t$. The omitted argument of the Step constructor (denoted by _) is a proof of $\rho_1 \in \mathcal{R}$. Note that x is the variable that is used in both rewrite rules.

```
Definition σ t (y : X) : term :=
  match beq_var y x with
  | true ⇒ t
  | false ⇒ Var y
  end.
Lemma fact_π^source : ∀ (n m : nat) (t : term),
  (U^n D^m □)[substitute (σ (U^m t)) (lhs ρ_1)] ↔ U^n D^{S m} U^{S m} t.
Lemma fact_π^target : ∀ (n m : nat) (t : term),
  (U^n D^m □)[substitute (σ (U^m t)) (rhs ρ_1)] ↔ U^n D^m U^m t.
Definition π n m t : U^n D^{S m} U^{S m} t →_R U^n D^m U^m t :=
  Step ρ_1 (U^n D^m □) (σ (U^m t)) _ (fact_π^source n m t) (fact_π^target n m t).
```

Generalising these rewrite steps π, we construct the rewrite sequences φ_a. In their recursive definition, the snoc function is used to prepend $(\pi\ n\ m\ t)$ to $(\varphi_a\ n\ m\ t)$. Doing some arithmetic, we obtain that these rewrite sequences can be used to define rewrite sequences φ_b of a more useful type.

```
Fixpoint φ_a n m t : U^n D^m U^m t →>_R U^n t :=
  match m with
  | O ⇒ Nil (U^n t)
  | S m ⇒ snoc (π n m t) (φ_a n m t)
  end.
Definition φ_b n : U^n (υ' n) →>_R U^{S n} (υ' (S n)) :=
  φ_a n (S (2 × n)) (U @ υ' (S n)).
```

We concatenate all rewrite sequences φ_b to construct rewrite sequences from υ to a term that is equal to U^ω up to any given depth.

```
Fixpoint φc n : υ →↠R U^n (υ' n) :=
  match n with
  | O ⇒ Nil υ
  | S n ⇒ concat (φc n) (φb n)
  end.
```

The definition of the final rewrite sequence φ is done by combining φ_c with a proof that the target terms converge to U^ω.

```
Lemma conv_φc : converges (fun n ⇒ U^n (υ' n)) U^ω.
Definition φ : υ →↠R U^ω := Lim φc conv_φc.
Lemma wf_φ : wf φ.
```

We can prove $\upsilon \twoheadrightarrow D^\omega$ in a similar way and conclude by proving our main theorem.

```
Lemma no_un_R : ¬ unique_normal_forms R.
Theorem no_un_wo : ¬ ∀ F X R,
  weakly_orthogonal (F := F) (X := X) R → unique_normal_forms R.
```

Chapter 5

Discussion

In Chapter 3 we present our formalisation of infinitary term rewriting and in the current chapter we discuss some aspects of it. First, the representation we use for rewrite sequences is motivated and the resulting definitions are related to the notion of convergence. We then comment on some technicalities in the implementation. Finally, we conclude with an evaluation of our formalisation.

5.1 Representing Rewrite Sequences

In Definition 19, transfinite rewrite sequences are introduced as partial functions from ordinals to rewrite steps, with the appropriate conditions on source and target terms of subsequent rewrite steps. Could we not translate this directly to Coq?

A problem with partial functions from ordinals to rewrite steps is that we would need a decidable order on the ordinals. Consider a non-trivial rewrite sequence of length α. The partial function representing this rewrite sequence must compare its input value (an ordinal) to all possible input values (ordinals up to α) in order to decide what rewrite step to produce, in finite time.

The order on our tree ordinals is not decidable (consider comparing the upper bounds of two infinite sequences). This could be remedied by using a different representation for the ordinals, for example axiomatically, as Cantor normal

forms, or as sets. We feel that the inductively defined tree ordinals are a more natural representation in the constructive type theory of Coq.

Comparison of ordinals up to some given upper bound may be decidable, so another remedy for this problem would be to only consider rewrite sequences of limited length. Motivated by the Compression Lemma, we could go even further and restrict our representation to rewrite sequences of length $\leq \omega$. This would severely cripple our formalisation, since much of the theory of infinitary rewriting could not be developed with this representation (e.g. the Compression Lemma itself).

Another argument in favour of our representation based on tree ordinals is that it seems natural for a Coq formalisation. As a comparison, lists of finite length are usually defined by induction in Coq, not as partial functions from the natural numbers. Our representation can be seen as a generalisation of inductively defined finite lists to lists of transfinite length.

5.2 Convergence of Rewrite Sequences

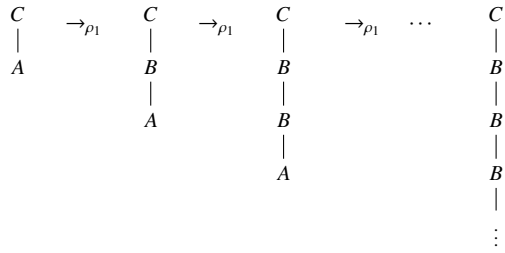
The inductively defined rewrite sequences from Section 3.3 are not necessarily (weakly) convergent. A rewrite sequence of limit length satisfies the condition that the target terms of the Lim branches converge but this is too weak to establish convergence of the rewrite sequence itself.

The depths of the rewrite steps are not considered at all in our formalisation and therefore it obviously does not implement strong convergence. Furthermore, the discussion in this section also applies to the notions of continuity.

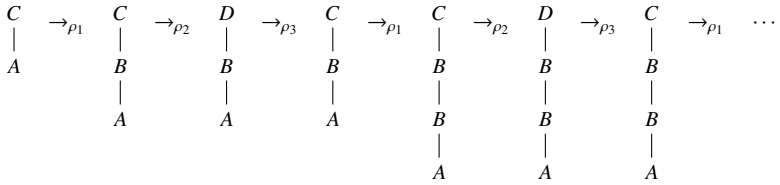
We consider an example of a rewrite sequence that satisfies our inductive definition from Section 3.3 but is not weakly convergent. Let A be a constant and B, C, D unary function symbols. We use the following three rewrite rules:

$$\rho_1 : A \to B(A) \qquad \rho_2 : C(x) \to D(x) \qquad \rho_3 : D(x) \to C(x)$$

The term $C(A)$ rewrites in ω many ρ_1-steps to $C(B^\omega)$.



We modify this rewrite sequence such that in between every two ρ_1-steps, the root symbol C is changed to D and back to C. The resulting rewrite sequence does not have a limit and is not weakly convergent.



We can define this rewrite sequence as the limit of $(\varphi_n)_{n\in\mathbb{N}}$, where $+\!\!+$ denotes concatenation of rewrite sequences:

$$\varphi_0 \;:\; C(A) \to^0 C(A)$$
$$\varphi_{n+1} \;:\; \varphi_n +\!\!+ C(B^n(A)) \to_{\rho_1} C(B^{n+1}(A)) \to_{\rho_2} D(B^{n+1}(A)) \to_{\rho_3} C(B^{n+1}(A))$$

The target terms $C(B^{n+1}(A))$ converge to $C(B^\omega)$ and this construction can thus be used with our inductive definition of rewrite sequences, where we take φ_n to be the n^{th} branch of the Lim constructor.

It is not clear to us whether there is some natural translation of the convergence conditions to our formalisation. For completeness we include a (not so natural) translation of convergence, but we were not able to use it in our development. Even proving the simplest convergent rewrite sequences to satisfy these definitions seems too involved.

```
Fixpoint weakly_convergent s t (φ : s ↠ℛ t) : Prop :=
  match φ with
  | Nil _ ⇒ True
  | Cons _ _ ψ _ _ ⇒ weakly_convergent ψ
  | Lim _ _ f t _ ⇒ (∀ n, weakly_convergent (f n)) ∧ ∀ d, ∃ ι, ∀ κ,
                      φ[ι]^SEQ ⊑ φ[κ]^SEQ → φ[κ]^L ≡≤d t
  end.
```

```
Fixpoint strongly_convergent s t (φ : s ↠_R t) : Prop :=
  match φ with
  | Nil _ ⇒ True
  | Cons _ _ ψ _ _ ⇒ strongly_convergent ψ
  | Lim _ _ f t _ ⇒ (∀ n, strongly_convergent (f n)) ∧
    ∀ d, ∃ ι, ∀ κ,
      φ[ι]^SEQ ⊑ φ[κ]^SEQ → d ≤ depth φ[κ]^STP
  end.
```

5.3 Design Decisions

We describe a number of design decisions that we had to make during implementation.

5.3.1 Coinductive Terms

We define the type of infinite terms by coinduction. Another often used definition of Ter^{∞}_{Σ} is by partial functions from positions to symbols. This definition is troublesome to translate to Coq, which admits no partial functions. It might be possible to circumvent this restriction using option types, or by some other means, but we suspect that the resulting notion of terms would not be elegant. Coinduction is a native concept in Coq and we use it as such. The consequence is that, in defining infinite terms, we are restricted to definitions in guarded form, see Section A.4.

5.3.2 Positions

The set of positions of a given term is a restricted list of natural numbers. This could probably be translated to Coq by encoding the restriction in the type of positions. It would result in quite complicated definitions, however, and therefore we choose for subterm and dig to take any list of natural numbers and return option types to distinguish between existing and non-existing positions. This comes at the cost of having to handle the option types with every call of subterm or dig.

5.3.3 Bisimilarity in Rewrite Steps

In the definition of rewrite steps (Section 3.3), we include bisimilarity conditions on the source and target terms. The reason is that we cannot rely on Coq's small standard equality. For consider two rewrite steps $s_1 \to s_2$ and $t_1 \to t_2$, where $s_2 \leftrightarrows t_1$. We should be able to construct the rewrite sequence consisting of these two steps, but the fact that the target term of the first and the source term of the second can be identified must be encoded somewhere. There are two places to do this: in the Cons constructor for rewrite sequences, or in the definition of rewrite steps. We choose the latter because it turns out to be technically more elegant.

5.3.4 One-Hole Contexts

We know of two obvious ways to represent multi-hole contexts: by the coinductive term datatype augmented with a constructor for holes or by a function whose arguments represent holes. Both are unsatisfactory. The first representation requires a dynamic check on the number of holes and it admits contexts with infinitely many holes. The second representation cannot guarantee that each hole occurs exactly once and its type, encoding the number of holes, cannot be generalised. Lindley (2008) presents a quite involved solution based on a type-level difference encoding of natural numbers. We bypass the problem by only considering one-hole contexts.

5.4 Conclusions

Our representation of transfinite rewrite sequences based on tree ordinals is original work. While the representation is natural to implement in Coq, we are disappointed by not being able to come up with a natural translation of convergence for this representation. Without such a translation, we feel our definitions are not satisfactory and we hope that future work can remedy this.

In an alternative representation as partial functions from ordinals to rewrite steps, convergence can be expresssed directly, but it has other problems. However, it seems worthwhile to also investigate this representation further, for example using the ordinals in Cantor normal form by Castéran (2006). Jeroen

Ketema showed by personal communication that, assuming some decidable properties on the ordinal representation, constructive formalisation of some of the infinitary rewriting theory is possible.

Working with Coq has been an overall pleasant experience. During this development, our main disturbance was with the guardedness condition on corecursive definitions. Sometimes unguarded but natural definitions are easily seen to be productive yet not accepted by Coq. One could hope for future additions to Coq in this area, for example in the form of user-supplied productivity proofs for corecursive definitions to be admitted.

Still, the amount of work that is required for a formalisation like this is disproportionally large. Especially the implementation of the counterexample to UN^∞ in weakly orthogonal systems from Chapter 4 takes only a page to formulate on paper, while it took us some 2,000 lines of Coq code to formalise.

The complete formalisation consists of some 6,500 physical source lines of code (LOC), roughly distributed as follows:

LOGICAL MODULE	LOC
coinductive terms	1,250
tree ordinals	900
transfinite rewriting	1,500
UN^∞ and weak orthogonality	2,000
total	$\approx$ 6,500

Appendix A

The Coq Proof Assistant

Coq (The Coq development team, 2009) is based on the formal language *Calculus of Inductive Constructions* (Coquand and Huet, 1988; Paulin-Mohring, 1993), which is essentially a typed λ-calculus with inductive types. In this language, logical propositions are represented as types and proofs of such propositions are λ-terms, motivated by the Curry–Howard–de Bruijn correspondence (Girard, 1989). The core of the Coq system is its type checking algorithm.

We present a very short introduction to Coq and refer to Bertot and Castéran (2004) and Chlipala (2009) for further reading. Sections A.4 and A.3 discuss two technicallities related to the Coq development described in Chapter 3.

A.1 Types and Propositions

Every term in Coq has a type and every type is also a term. The type of a type is called a *sort* and the sorts in Coq are

- `Prop`, the sort of logical propositions,
- `Set`, the sort of program specifications and datatypes,
- $\texttt{Type}_0$, the sort of `Prop` and `Set`, and
- $\texttt{Type}_{i+1}$, the sort of $\texttt{Type}_i$.[1]

[1] The subscripts i of the sorts $\texttt{Type}_i$ are invisible to the user and only used internally.

For example, nat is the datatype of the natural numbers. It lives in Set and is defined inductively.

```
Inductive nat : Set :=
  | O : nat
  | S : nat → nat.
```

The logical proposition that for every natural number n, there exists a natural number m larger than n, can be stated as a term of sort Prop.

```
(∀ n : nat, ∃ m : nat, n < m) : Prop
```

Using the vocabulary of types and terms, the universal quantifier ∀ is called the *product type constructor*. A product type $\forall\ x : T,\ U$ is called *dependent* if x occurs free in U, otherwise it is written $T \to U$. The type of the constructor symbol S defined above, for example, is that of functions from nat to nat and is not dependent. The non-dependent function space notation $T \to U$ is also used for logical implication, justified by the Curry–Howard–de Bruijn correspondence.

A.2 Terms and Proofs

Proofs of logical propositions can be defined in two ways. First, we can write a proof term directly. The only requirement is that this term has as type the logical proposition that we want to prove (again, justified by the Curry–Howard–de Bruijn correspondence). Second, we can use *tactics* to construct a proof term interactively, in a way mimicking natural deduction.

As an example of the use of tactics, we prove the proposition from the previous section. This is done by stating the proposition, after which the system enters a goal-directed proof mode. In this mode, we are presented with a goal, consisting of (i) a context of local variables that are available (ii) a proposition denoting what remains to be proven. Tactics can now be applied to progressively transform the goal into a simpler goal. When the goal is simple enough to be solved directly by applying a tactic, we are done proving the proposition.

```
Lemma lt_serial : ∀ n : nat, ∃ m : nat, n < m.
Proof.
  intro n.  exists (S n).  auto.
Qed.
```

In this example, we use the tactic `intro` to introduce the variable n to the context. With `exists`, we supply a witness for the existential quantification. At this point, the goal is simple enough to be solved directly by the `auto` tactic.

Recursive functions are defined using the `Fixpoint` keyword. The function must have an argument of an inductive type that is structurally decreasing with each recursive call. Consider for example the definition of the factorial function, which also shows how a case analysis on values of inductive types can be done with the `match` keyword.

```
Fixpoint factorial (n : nat) : nat :=
  match n with
  | O ⇒ S O
  | S n ⇒ S n × (factorial n)
  end.
```

A.3 The Positivity Condition

Coq restricts inductive definitions to those that satisfy the *positivity condition*. The reason for this is that definitions that fail this (syntactic) criterion may lead to an inconsistent system. For a precise definition of positivity, consult The Coq Reference Manual, Section 4.5.3.

Consider again the definition of rewrite sequences from Section 3.3. A more natural way to define the type of the Lim constructor might be by using a Σ-type instead of a separate function for the target terms of the branches.

```
| Lim : ∀ s t (f : nat → { t' : term & s →ℛ t' }),
          converges (fun n ⇒ projT1 (f n)) t → (s →ℛ t)
```

However, this type definition does not satisfy the positivity condition and therefore we cannot use it. We feel that the definition from Section 3.3, which does satisfy the condition, models our intentions adequately.

A.4 Guardedness in Corecursive Definitions

In Coq, coinductive types can be defined using the `CoInductive` keyword (Giménez and Castéran, 2007). No induction principles are defined for these

types, because they are not necessarily well-founded.[1] Objects in a coinductive type may be infinite (i.e. contain an infinite amount of constructors). However, in order to guarantee productivity, definitions of such objects are required by Coq to be in *guarded* form (Giménez, 1994). A corecursive definition in guarded form satisfies two (syntactical) conditions. First, every corecursive call must occur inside at least one constructor (of the same coinductive type). Second, every corecursive call may only occur inside abstractions or constructors (of the same coinductive type).[2]

In the term definition, we use a vector type, parameterised by the type of its element and its size. Naturally, one would implement a vector type in Coq inductively, as for example has been done in the standard library.

```
Inductive vector (A : Type) : nat → Type :=
  | Vnil : vector A 0
  | Vcons : A → ∀ n, vector A n → vector A (S n).
```

Now consider the following trivial example of a basic operation on terms by corecursive traversal.

```
CoFixpoint id (t : term) : term :=
  match t with
  | Var x ⇒ Var x
  | Fun f args ⇒ Fun f (vmap id args)
  end.
```

This definition is ill-formed, since the corecursive call to id is not guarded.[3] We define a recursive type of vectors as an alternative to the inductive type.

```
Inductive Fin : nat → Type :=
  | First : ∀ n, Fin (S n)
  | Next : ∀ n, Fin n → Fin (S n).
Definition vector (A : Type) (n : nat) := Fin n → A.
```

This makes for a definition of vmap that is just an abstraction, and therefore solves the guardedness problem in id.

```
Definition vmap A B (f : A → B) n : vector A n → vector B n :=
  fun v i ⇒ f (v i).
```

[1] Coq automatically derives induction principles for inductive definitions.
[2] To be more precise, the corecursive call is also allowed to occur inside `match` constructs and other corecursive definitions.
[3] The call to id is hidden inside vmap, which is defined by recursion on the vector *args*.

References

Kenneth Appel and Wolfgang Haken. Every map is four colourable. In *Bulletin of the American Mathematical Society*, volume 82, pages 711–712. 1976.

Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer-Verlag, 2004.

Frédéric Blanqui and Adam Koprowski. CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. Under consideration for publication in Mathematical Structures in Computer Science, `http://www-rocq.inria.fr/~blanqui/color-pdf.html`, 2010.

Georg Cantor. *Contributions to the Founding of the Theory of Transfinite Numbers*. The Open Court Series of Classics of Science and Philosophy. The Open Court, 1915. Translated by Philip Jourdain; reprinted by Dover, 1955.

Pierre Castéran. Ordinal notations based on cantor and veblen normal forms. `http://coq.inria.fr/contribs/Cantor.html`, 2006.

Adam Chlipala. *Certified Programming with Dependent Types*. 2009. Draft textbook, `http://adam.chlipala.net/cpdt/`.

Évelyne Contejean, Pierre Courtieu, Julien Forest, Olivier Pons, and Xavier Urbain. Certification of automated termination proofs. In B. Konev and F. Wolter, editors, *FroCos '07*, volume 4720 of *Lecture Notes in Artificial Intelligence*, pages 148–162. Springer, 2007.

Thierry Coquand and Gerard Huet. The calculus of constructions. *Information and Computation*, 76(2-3):95–120, 1988.

E.C. Dennis-Jones and S.S. Wainer. Subrecursive hierarchies via direct limits. In M. Richter, E. Börger, W. Oberschelp, B. Schinzel, and W. Thomas, editors, *Computation and Proof Theory*, volume 1104 of *Lecture Notes in Mathematics*, pages 117–128. Springer, 1984.

Jörg Endrullis, Clemens Grabmayer, Dimitri Hendriks, Jan Willem Klop, and Vincent van Oostrom. Unique normal forms in infinitary weakly orthogonal rewriting. In C. Lynch, editor, *RTA '10*, volume 6 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 85–102. Schloss Dagstuhl, 2010.

Eduardo Giménez. Codifying guarded definitions with recursive schemes. In *TYPES '94*, volume 996 of *Lecture Notes in Computer Science*, pages 39–59. Springer-Verlag, 1994.

Eduardo Giménez and Pierre Castéran. A tutorial on [co-]inductive types in Coq. `http://coq.inria.fr/documentation`, 2007.

Jean-Yves Girard. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989. Translated and with appendices by Paul Taylor and Yves Lafont.

Georges Gonthier. A computer-checked proof of the four colour theorem. `http://research.microsoft.com/~gonthier/4colproof.pdf`, 2005.

Thomas C. Hales, John Harrison, Sean McLaughlin, Tobias Nipkow, Steven Obua, and Roland Zumkeller. A revision of the proof of the Kepler conjecture. *Discrete and Computational Geometry*, 44:1–34, 2009.

Peter Hancock. (Ordinal-theoretic) proof theory. Course notes from Midlands Graduate School, 2008.

Richard Kennaway, Jan Willem Klop, Ronan Sleep, and Fer-Jan de Vries. Transfinite reductions in orthogonal term rewriting systems. *Information and Computation*, 119:18–38, 1995.

Jan Willem Klop and Roel de Vrijer. Infinitary normalization. In S. N. Artemov, H. Barringer, A. S. d'Avila Garcez, L. C. Lamb, and J. Woods, editors, *We Will Show Them: Essays in Honour of Dov Gabbay*, volume 2, pages 169–192. College Publications, 2005.

Edmund Landau. *Grundlagen der Analysis*. Chelsea Publishing Company, fourth edition, 1965. First edition 1930.

Sam Lindley. Many holes in Hindley–Milner. In *ML '08*, pages 59–68. ACM, 2008.

The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2009. Version 8.2, `http://coq.inria.fr/doc/`.

Paul-André Melliès. Braids described as an orthogonal rewriting system. Manuscript, `http://www.pps.jussieu.fr/~mellies/papers/braids.ps`, 1995.

Rob Nederpelt, Herman Geuvers, and Roel de Vrijer, editors. *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1994.

Christine Paulin-Mohring. Inductive definitions in the system Coq—rules and properties. In M. Bezem and J.F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, volume 664 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.

Neil Robertson, Daniel Sanders, Paul Seymour, and Robin Thomas. The four-colour theorem. *Journal of Combinatorial Theory, Series B*, 70:2–44, 1997.

Terese. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

Freek Wiedijk. Formal proof—getting started. *Notices of the American Mathematical Society*, 55(11):1408–1414, 2008.