Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Formal System Analysis Research Group

Automatically proving equality of infinite sequences

Master's Thesis

Stan Roelofs

Supervisors:
Hans Zantema

Version 1.1

Eindhoven, October 2019

Abstract

First order inductive theorem proving deals with proving new equations called *goals* based on a given set of equations called *axioms*. More specifically, we are interested in proving that the axioms logically imply the goals. Here the intended notion of equality is called *ground convertibility*. To prove that two terms are ground convertible we typically need induction.

In this paper we will investigate how we can automate these proofs. This is done by describing various induction techniques which are suitable for automation. The equations can either describe finite or infinite terms. In this paper we are mainly interested in applying these ideas of induction on infinite objects. The focus is on the most simple form of infinite objects: infinite sequences of some data type. The induction techniques cannot be applied on such infinite objects by default. However using a special operator `take` we can also apply our induction techniques on infinite data. We show that applying induction on infinite data yields a technique which is equivalent to *circular coinduction*.

Finally we describe how our techniques can be implemented. Throughout the paper we describe several non-trivial examples of equations that can be proved. These proofs are generated automatically by our tool, showing that our induction techniques are indeed suitable for automation.

Keywords: equality, automatic, induction, infinite, proving

Contents

Chapter 1

Introduction

First order theorem proving deals with proving new equations based on a given set of equations. The given set of equations are called *axioms*, and the new equations that should be proven are called *goals*. The question that we are interested in and would like to answer is: do the axioms logically imply the goals? Inductive theorem provers try to solve this question in an automated way.

To demonstrate the kinds of problems we would like to solve, consider the following example involving addition over the natural numbers defined by 0 and s (successor). With these two functions we can create any natural number: 0 corresponds to 0, 1 corresponds to $s(0)$, 2 corresponds to $s(s(0))$, and so on. We can now define addition over the natural numbers with the + symbol and the following two equations: $0 + x = x$ and $s(x) + y = s(x + y)$. As the goal we would like to prove that $x + 0 = x$. We would expect this to be true, however we cannot prove this goal, since we cannot get rid of the 0 in $x + 0$ by only applying the given equations.

However in this context the intended notion of equality is typically the weaker notion of *ground convertibility*. This means that if we replace all of the variables in the equation by *ground terms*, then the two sides of the equation can always be converted to each other by only applying the given equations. Ground terms are terms not containing any variables. In this example these ground terms are simply the natural numbers composed from the constructors 0 and s. Hence, if $x + 0$ and x are ground convertible, then replacing x by any term t constructed from 0 and s means that $t + 0$ can be converted to t by applying the given equations.

In inductive theorem proving, we prove ground convertibility of such an equation by *induction*. If the equations are designed in such a way that every ground term is convertible to a ground constructor term, it is correct to do induction on the set of constructors. In Chapter 3 we will show how the given example can be proved using induction.

Several techniques have been developed to do this inductive theorem proving efficiently. Many of the existing automated methods for inductive theorem proving are based on the Boyer-Moore approach. This was first presented in the book *A Computational Logic* by Boyer and Moore [5, 19, 15].

Rippling [6, 3] is another technique used in automated theorem proving systems. Rippling is a heuristic that controls the search in inductive proofs. This is achieved by using a more restricted form of rewriting where annotations are used to guide the proof. The key idea is to manipulate the induction conclusion to be able to use the induction hypothesis in its proof.

Inductive theorem proving often requires suitable lemmas to help with the inductive proof. These need to be manually provided to the system. Identifying such lemmas is a major challenge and requires human expertise. Automatically searching for relevant lemmas has been investigated as a solution, for example in [7, 9]. A more recent approach uses machine learning techniques to search for lemmas automatically [16].

Like most of the research done in this area, these techniques are focused on the setting where

terms are finite. Although we will discuss inductive theorem proving for finite terms in Chapter 3, in this paper we are mostly interested in applying these induction techniques on infinite data types, such as infinite sequences. Different techniques for automatically proving equalities have been developed for both of these settings.

For automatically proving equality of infinite objects the *Circular Coinduction* technique has been developed [20, 12, 17, 11, 13, 24]. The name circular coinduction is based on how it operates: it searches for periodic behaviors of the terms to prove equivalent. More specifically it systematically explores the behaviors of the property to prove until each path results in a truth or a cycle. Informally its correctness can be argued as follows: each derived path results in a truth or a cycle, which means that there is no way to show the two original terms are different, therefore they must be equivalent.

Circular coinduction can be formalized as a proof system consisting of three rules that derives pairs of the form $H \models G$. Here H (hypotheses) and G (goals) are sets of equations. In circular coinduction some equations can be *frozen*. This means that the equation can be applied at the top level, but otherwise it cannot be used in equational reasoning. This is necessary to restrict the application of the hypothesis, without freezing the conclusion would vacuously hold by equational reasoning.

The first rule says that as soon as there are no goals left, the derivation is finished. The second rule discards the goals that can be proven by equational reasoning using the equations in H. The third rule is the circular coinduction rule. This rule says that in order to prove the equation e, we assume that it holds (by adding it to H) and prove its derivatives $\Delta[e]$. Intuitively the idea behind this is: "if one cannot show it wrong then it is right" [20].

This technique was implemented in the tool CIRC [18], an extension of the rewriting engine *Maude* [8]. CIRC successfully proves several interesting examples, however since it is based on a rewriting engine the equations can only be used in one direction. This can be problematic, as some examples require the equations to be used in both directions. The tool *Streambox* [24] was developed which does not have this limitation and outperforms CIRC for several interesting equalities.

The basic circular coinduction proof system was later extended by allowing the use of *special contexts* to generate additional hypotheses. This results in a modified and more powerful proof system.

The extension with special contexts changes the third rule: rather than assuming that e holds, we are allowed to assume $C[e]$ holds for any special context C. A context refers to a term with a *hole*, which can be replaced by a different term of the same sort. This yields a new term where the old term is now a subterm surrounded by the context, which explains why it is called a context. Special contexts are contexts that satisfy certain constraints, that allow them to be used to create extra hypotheses of the shape $C[l] = C[r]$ for the goal $l = r$. The use of special contexts was implemented in CIRC and using this extended proof system more examples can be proved.

Whereas most older work focuses on the setting where terms are finite, in this paper we will focus on applying induction on infinite data, such as infinite sequences. As discussed the technique of *circular coinduction* has been developed to automatically prove equality of infinite objects specified by equations. We will prove equality of infinite objects using a different approach based on the techniques described in a recent paper [23]: to prove that the objects t, u of the same sort are equal, we prove that $\mathsf{take}(x, t) = \mathsf{take}(x, u)$ for every natural number n, where $\mathsf{take}(n, a)$ just takes the n-th element of the object a. With this approach we can use the same induction methods for finite and infinite objects. Furthermore we will show that this approach is equivalent to circular coinduction.

For the methods described in this paper we do not require properties that are typically necessary in inductive theorem proving techniques. For example often it is required that the set of equations are designed such that applied from left to right they form a terminating rewrite system, or two distinct ground terms are not allowed to be convertible. By not requiring these properties we can find applications outside this standard format. For example we can prove equations on integers, where two ground terms can be convertible to each other.

We will introduce a new kind of induction for proving equality which is related to strong induction. This allows us to prove some examples on which the previous methods fail. This method allows us to use extra hypotheses in certain cases.

Whereas most of the other work for the setting with infinite data focuses solely on one-sided infinite sequences, we will briefly consider different kinds of infinite objects and show how our methods can also be applied on those objects.

Our work is not just theoretical, we also investigate how these induction techniques can be automated. The methods described in this paper have been implemented in a tool. This tool can prove most of the examples found in this thesis fully automatically, including some non-trivial examples which require a fairly extensive proof. The most common reason it fails is that some proofs require auxiliary lemmas (equations). To solve this problem the tool is capable of searching and discovering those lemmas automatically.

This document is organized as follows. In Chapter 2 we give some preliminaries including the notion of ground convertibility. We will introduce several induction variants for the setting with finite terms in Chapter 3. In Chapter 4 we will show how we can apply these induction techniques from Chapter 3 on infinite objects, such as infinite sequences. We will consider some non-trivial examples of equalities we can prove, for instance equalities on *morphic sequences* or two-sided infinite sequences. In Chapter 5 we will describe some details about the implementation of our tool, which is capable of proving most of the examples found in this paper fully automatically. We conclude in Chapter 6.

Chapter 2

Preliminaries

Let S be a finite set of sorts. An S-sorted set A consists of a set A_s for every sort $s \in S$.

Define the S-sorted signature Σ to consists of a set of *function symbols*. Each function symbol f has a number of inputs $n \geq 0$ which is called the *arity* of f, input sorts $s_1, \ldots, s_n \in S$ and an output sort s. This is described by the notation $f : s_1, \ldots, s_n \to s$, which will be commonly used throughout this document. We refer to the output sort s of a function f as simply the sort of f. We call f a constant of sort s if the arity of f is 0. In general we will use standard function notation throughout this document, i.e. we write $f(t_1, \ldots, t_n)$ for the function f applied on the arguments $t_1, \ldots, t_n$. In some cases we may diverge from this and use infix notation where this seems more appropriate, for example we may write $x + y$ instead of $+(x, y)$ since the former is more familiar.

Let $\mathcal{X}$ be an S-sorted set of variables. A term over $\Sigma, \mathcal{X}$ is constructed from the functions in Σ and the variables in $\mathcal{X}$. We call a term a *ground term* if it contains no variables.

The S-sorted set of terms $\mathcal{T}(\Sigma, \mathcal{X})$ over $\Sigma, \mathcal{X}$ is defined inductively as follows:

- If $x \in \mathcal{X}_s$ then $x \in \mathcal{T}(\Sigma, \mathcal{X})_s$.
- If $f : s_1, \ldots, s_n \to s$ and $t_i \in \mathcal{T}(\Sigma, \mathcal{X})_{s_i}$ for $i = 1, \ldots, n$ then $f(t_1, \ldots, t_n) \in \mathcal{T}(\Sigma, \mathcal{X})_s$.

In other words, all variables in $\mathcal{X}$ are in $\mathcal{T}(\Sigma, \mathcal{X})$. All constants are in $\mathcal{T}(\Sigma, \mathcal{X})$, and all terms that can be constructed using the functions in Σ and these variables and constants are in $\mathcal{T}(\Sigma, \mathcal{X})$.

The S-sorted set of ground terms $\mathcal{T}(\Sigma, \emptyset)$ is abbreviated to $\mathcal{T}(\Sigma)$.

A *well-founded relation* [14] R on a set of terms $\mathcal{T}(\Sigma, \mathcal{X})$ is a binary relation such that every non-empty subset S of terms has a minimal element with respect to R. A minimal element is an element m not related by sRm for any $s \in S$. In other words, a relation R is well-founded if:

$$\forall S \subseteq \mathcal{T}(\Sigma, \mathcal{X}) : (S \neq \emptyset \implies \exists m \in S, \forall s \in S : (\neg(sRm)))$$

Equivalently, a relation R is well-founded if it contains no *infinite descending chains*: that is, there exists no sequence $x_0, x_1, \ldots$ of elements such that $x_{n+1}\ R\ x_n$ for all natural numbers n.

A context $\mathbf{C}$ of sort s' is a term which has one occurrence of a distinguished variable $*$ of a sort s called a *hole*. If t is a term of sort s, then $\mathbf{C}[t]$ is the term of sort s' obtained by replacing $*$ by t in $\mathbf{C}$.

An equation over Σ of sort s is a pair of terms $(t, u) \in \mathcal{T}(\Sigma, \mathcal{X})^2_s$. Rather than writing (t, u) we will use the notation $t = u$ for equations.

A *substitution* $\sigma : \mathcal{X} \to \mathcal{T}(\Sigma, \mathcal{X})$ maps every variable $x \in \mathcal{X}$ to a term $\sigma(x)$ in $\mathcal{T}(\Sigma, \mathcal{X})$ of the same sort. The term $t\sigma \in \mathcal{T}(\Sigma, \mathcal{X})$ is obtained by replacing every variable x in t by $\sigma(x)$ for $t \in \mathcal{T}(\Sigma, \mathcal{X})$ and $\sigma : \mathcal{X} \to \mathcal{T}(\Sigma, \mathcal{X})$. If σ maps every variable to a ground term, i.e. $\sigma : \mathcal{X} \to \mathcal{T}(\Sigma)$, then σ is called a *ground substitution*. We write $t[x_1 := u_1, \ldots, x_n := u_n]$ for the term $t\sigma$ obtained from a term t and a substitution σ defined by $\sigma(x_1) = u_1, \ldots, \sigma(x_n) = u_n$ and $\sigma(y) = y$ for all

$y \in \mathcal{X} \setminus \{x_1, \ldots, x_n\}$.

The rewrite relation $\to_\mathcal{E}$ for a set of equations $\mathcal{E}$ over Σ is defined inductively as follows:

- If $t = u \in \mathcal{E}$ and σ is a substitution $\mathcal{X} \to \mathcal{T}(\Sigma, \mathcal{X})$ then $t\sigma \to_\mathcal{E} u\sigma$.
- If $f : s_1, \ldots, s_n \to s$, and $t_i \to_\mathcal{E} u_i$ for some i, and $t_j = u_j$ for all $j \neq i$, then $f(t_1, \ldots, t_n) \to_\mathcal{E} f(u_1, \ldots, u_n)$.

As described in [22] a *term rewrite system* (TRS) consists of a set of equations which should only be used in one direction. Since the equations are only allowed to be applied in one direction, they are often called *rewrite rules* instead of equations, and we write $l \to r$ instead of $l = r$.

A term t is called a *normal form* if no u exists such that $t \to_\mathcal{E} u$. A term t is called a normal form of u if t is a normal form and u rewrites to t in zero or more steps, i.e. $u \to_\mathcal{E}^* t$. Where $\to_\mathcal{E}^*$ is the reflexive, transitive closure of $\to_\mathcal{E}$. Without extra requirements a term can have no normal form, or more than one normal form. There are some important properties a term rewrite systems R can have:

- Weakly Normalizing (WN): Every term t has a normal form.
- Strongly Normalizing (SN): There exists no infinite sequence of terms $t_1, t_2, \ldots$ such that $t_i \to_R t_{i+1}$ for all $i \geq 1$. In other words, we cannot rewrite any term infinitely many times. If R is strongly normalizing we may also call it *terminating*.
- Confluence (Church-Rosser, CR): If $t \to_R^* u$ and $t \to_R^* v$ then there exists a term w such that $u \to_R^* w$ and $v \to_R^* w$.
- Local (weak) confluence (Weak Church-Rosser, WCR): If $t \to_R u$ and $t \to_R v$ then there exists a term w such that $u \to_R^* w$ and $v \to_R^* w$.

In general determining whether a TRS is terminating is undecidable. This can be shown by transforming an arbitrary TRS into a Turing machine such that the TRS is terminating if and only if the Turing machine halts on every initial configuration. However, a valid technique exists that can be applied in many cases. The general idea is to assign a weight function to each rewrite rule $l \to r$ such that the weight of l is greater than the weight of r. This weight function must be monotonic, meaning that if for all $a_i, b_i \in \mathbb{N}$ for $i = 1, \ldots, n$ with $a_i > b_i$ for some i and $a_j = b_j$ for $i \neq j$ then $f(a_1, \ldots, a_n) > f(b_1, \ldots, b_n)$.
For a map $\alpha : \mathcal{X} \to \mathbb{N}$ the weight function $[\cdot, \alpha]$ is defined inductively by:

$$[x, \alpha] = \alpha(x)$$

$$f(t_1, \ldots, t_n), \alpha] = [f]([t_1, \alpha], \ldots, [t_n, \alpha])$$

Theorem 1 (Termination rewrite system [22])
Let R be a TRS over a set of symbols Σ. For every symbol $f \in \Sigma$ let $[f]$ denote the weight function assigned to the symbol f. Assume that:

- *$[f]$ is monotonic for every symbol $f \in \Sigma$, and*
- *$[l, \alpha] > [r, \alpha]$ for every $\alpha : \mathcal{X} \to \mathbb{N}$ and every rule $l \to r$ in R*

Then R is terminating.

Termination of a TRS can be checked automatically using tools like AProVE [10] which can exploit more powerful techniques.

The symmetric closure of the rewrite relation $\to_\mathcal{E}$ is $\leftrightarrow_\mathcal{E}$. An alternative way to define $\leftrightarrow_\mathcal{E}$ is as follows: $t \leftrightarrow_\mathcal{E} u$ if and only if there exists $l = r \in \mathcal{E}$, a context C and a ground substitution τ such that $t = C[l]\tau \wedge u = C[r]\tau$ or $t = C[r]\tau \wedge u = C[l]\tau$.

$\sim_\mathcal{E}$ is the reflexive transitive closure of $\leftrightarrow_\mathcal{E}$. Therefore $t \sim_\mathcal{E} u$ if and only if there exist terms $t_0, \ldots, t_n$ all of the same sort (for $n \geq 0$) such that $t = t_0$ and $u = t_n$ and for all $i = 1, \ldots, n$ $t_i - 1 \leftrightarrow_\mathcal{E} t_i$. If t and u are two terms such that $t \sim_\mathcal{E} u$ then t and u are called *convertible*.

Definition 1 (Ground convertibility)
Two terms t, u of the same sort are ground convertible if $t\sigma \sim_{\mathcal{E}} u\sigma$ for every ground substitution $\mathcal{X} \to \mathcal{T}(\Sigma)$. Ground convertibility of t and u is denoted by $t =_{\mathcal{E}} u$.

As described in the introduction ground convertibility is often the intended notion of equality of two terms. If two terms t and u are convertible then by definition they are also ground convertible. However, in general if t and u are ground convertible they are not necessarily convertible. This is shown by the example in the introduction. To prove that two terms are ground convertible we typically need induction.

Chapter 3

Induction

The main technique that we use to prove ground convertibility of two terms is induction. The following theorem based on Theorem 2 in [23] describes this main induction principle, which allows us to prove for terms t, u and a set of equations $\mathcal{E}$ that $t =_{\mathcal{E}} u$.

Theorem 2 (Main induction principle)
Let $\mathcal{E}$ be a set of equations over Σ. Let $C \subseteq \Sigma$ such that every $f \in C$ has the same sort s. Assume that for all $t \in \mathcal{T}(\Sigma)_s$ there exists $u \in \mathcal{T}(C)_s$ such that $u \sim_{\mathcal{E}} t$.

Let $t, u \in \mathcal{T}(\Sigma, \mathcal{X})$ be two terms of the same sort. Let $x \in \mathcal{X}_s$. For every $f : s_1, \ldots, s_n \to s \in C$ assume that

$$t[x := f(a_1, \ldots, a_n)] =_{\mathcal{E}'} u[x := f(a_1, \ldots, a_n)],$$

where $a_1, \ldots, a_n$ are fresh constants of sorts $s_1, \ldots, s_n$, respectively, and

$$\mathcal{E}' = \mathcal{E} \cup \bigcup_{s_i = s} \{t[x := a_i] = u[x := a_i]\}.$$

Then $t =_{\mathcal{E}} u$.

Note that only functions in C of the same sort s as x are considered, which may be distinct from the sort of t and u. In other words, the induction variable may have a different sort than t and u. In case f has no arguments of sort s, then $\mathcal{E} = \mathcal{E}'$, this may be called the *base case* of the induction. The second case where $\mathcal{E} \neq \mathcal{E}'$ may be called the *inductive step*.

Proof. We have to prove that $t =_{\mathcal{E}} u$. By the definition of ground convertibility we therefore have to prove that $t\sigma \sim_{\mathcal{E}} u\sigma$ for all ground substitutions $\sigma : \mathcal{X} \to \mathcal{T}(\Sigma)$. It suffices to prove this for all ground substitutions $\sigma' : \mathcal{X} \to \mathcal{T}(C)$. This follows from the first assumption: for every term $t \in \mathcal{T}(\Sigma)$ there exists a term $u \in \mathcal{T}(C)$ such that $u \sim_{\mathcal{E}} t$. Define $\sigma' : \mathcal{X} \to \mathcal{T}(C)$ as $\sigma'(x) = u$ and $\sigma'(y) = y$ for $y \in \mathcal{X} \setminus \{x\}$. If $t\sigma' \sim_{\mathcal{E}} u\sigma'$ then $t\sigma \sim_{\mathcal{E}} t\sigma' \sim_{\mathcal{E}} u\sigma' \sim_{\mathcal{E}} u\sigma$ and thus we obtain our required goal. Hence we may restrict ourselves to the case where $\sigma(x) \in \mathcal{T}(C)$.

We will now prove $t[x := v] =_{\mathcal{E}} u[x := v]$ for $v \in \mathcal{T}(C)$ by induction on the size of v. By the definition of C we know that v is of sort s. Therefore, it is of the shape $f(v_1, \ldots, v_n)$ for a symbol $f \in C$ of sort s.

By the assumption we know

$$t[x := f(a_1, \ldots, a_n)] =_{\mathcal{E}'} u[x := f(a_1, \ldots, a_n)]$$

for fresh constants $a_1, \ldots, a_n$. Since these are fresh constants, we can replace them by $v_1, \ldots, v_n$ to obtain:

$$t[x := f(v_1, \ldots, v_n)] =_{\mathcal{E}''} u[x := f(v_1, \ldots, v_n)]$$

where $\mathcal{E}'' = \mathcal{E} \cup \bigcup_{s_i = s} \{t[x := v_i] = u[x := v_i]\}$.

Since the size of v_i is less than the size of $v = f(v_1, \ldots, v_n)$, by the induction hypothesis we obtain $t[x := v_i] =_{\mathcal{E}} u[x := v_i]$, for all i with $s_i = s$. Then $t =_{\mathcal{E}} u$ follows from Theorem 3 and $t[x := f(v_1, \ldots, v_n)] =_{\mathcal{E}''} u[x := f(v_1, \ldots, v_n)]$ by defining P as $E'' \setminus E$. □

Theorem 3 describes that $t =_{\mathcal{E}} u$ if and only if $t =_{\mathcal{E}'} u$ where $\mathcal{E}'$ consists of the original set of equations $\mathcal{E}$ and additional equations $a = b$ for a, b such that $a =_{\mathcal{E}} b$. In other words, Theorem 3 describes that if we can prove $a =_{\mathcal{E}} b$, we may add $a = b$ as an additional equation to $\mathcal{E}$ to obtain the set of equations $\mathcal{E}'$. Then we can prove $t =_{\mathcal{E}} u$ if and only if we can prove $t =_{\mathcal{E}'} u$, which means we are allowed to add additional equations as auxiliary lemmas to $\mathcal{E}$ to prove ground convertibility of two terms.

Theorem 3 (Using auxiliary lemmas)
Let $\mathcal{E}$ be a set of equations over Σ. Let P contain pairs of terms t, u of equal sort such that $t =_{\mathcal{E}} u$. Let a, b be two terms of equal sort.

Then $a =_{\mathcal{E}} b$ if and only if $a =_{\mathcal{E}'} b$ where $\mathcal{E}' = \mathcal{E} \cup \bigcup_{(t,u)\in P}\{t = u\}$.

Proof. Assume that $a =_{\mathcal{E}} b$.

Then by the definition of $=_{\mathcal{E}}$ there is some conversion $a\sigma = t_0\sigma \leftrightarrow_{\mathcal{E}} \ldots \leftrightarrow_{\mathcal{E}} t_n\sigma = b\sigma$ for all ground substitutions σ. Since $\mathcal{E}$ is a subset of $\mathcal{E}'$, the exact same conversion can be used to show that $a =_{\mathcal{E}'} b$.

Assume that $a =_{\mathcal{E}'} b$. Then by the definition of $=_{\mathcal{E}'}$ there is some conversion $a\sigma = t_0\sigma \leftrightarrow_{\mathcal{E}'} \ldots \leftrightarrow_{\mathcal{E}'} t_n\sigma = b\sigma$ for all ground substitutions σ. Let $\mathcal{E}''$ be the set of equations constructed from the pairs in P, so $\mathcal{E}'' = \bigcup_{(t,u)\in P}$. If nowhere in this conversion any of the equations in $\mathcal{E}''$ are used, then clearly we also have $a =_{\mathcal{E}} b$ with the same conversion steps.

Otherwise, since $t =_{\mathcal{E}} u$ by the assumption and the definition of $=_{\mathcal{E}}$ there is some conversion $t\sigma = t_0\sigma \leftrightarrow_{\mathcal{E}} \ldots \leftrightarrow_{\mathcal{E}} t_n\sigma = u\sigma$ for all ground substitutions σ.

We can now build a conversion such that $a =_{\mathcal{E}} b$ as follows. For every step in the conversion where an equation $t = u$ in $\mathcal{E}''$ is used, we replace the step by the conversion from t to u. Such a conversion must exist, since $t =_{\mathcal{E}} u$. Now we obtain a new conversion $a\sigma = t'_0\sigma \leftrightarrow_{\mathcal{E}} \ldots \leftrightarrow_{\mathcal{E}} t'_n\sigma = b\sigma$ for all ground substitutions σ that uses only equations in $\mathcal{E}$.

Therefore we also have $a =_{\mathcal{E}} b$. □

Using Theorem 2 we can now prove ground convertibility of two terms of the same sort using induction. This is demonstrated by the following example, which we described earlier in the introduction.

Example 1 (Natural numbers plus zero)
Let the set of equations $\mathcal{E}$ be defined as follows: $\mathcal{E} = \{0 + x = x, s(x) + y = s(x + y)\}$. We only have one sort natural, and $\Sigma = \{0, s, +\}$. Define $C = \{0, s\}$.

It is easy to observe that for all $t \in \mathcal{T}(\Sigma)$ there exists $u \in \mathcal{T}(C)$ such that $u =_{\mathcal{E}} t$, since rewriting a term containing $+$ eventually yields a term constructed only from 0 and s. For now we will simply assume that this is the case, later we will present a Theorem (5) that allows us to prove this if the set of equations $\mathcal{E}$ satisfies certain constraints.

The goal is to prove $x + 0 = x$, so in this case t and u in Theorem 2 are defined as $t = x + 0$ and $u = x$. Now we have to show that for every $f : s_1, \ldots, s_n \to s \in C$ the following holds: $t[x := f(a_1, \ldots, a_n)] =_{\mathcal{E}'} u[x := f(a_1, \ldots, a_n)]$ for fresh constants $a_1, \ldots, a_n$ and $\mathcal{E}' = \mathcal{E} \cup \bigcup_{s_i=s}\{t[x := a_i] = u[x := a_i]\}$. Since we defined $C = \{0, s\}$ this yields the following goals:

$$(x + 0)[x := 0] =_{\mathcal{E}} x[x := 0] \quad \textit{and} \quad (x + 0)[x := s(a)] =_{\mathcal{E}'} x[x := s(a)]$$

where $\mathcal{E}' = \mathcal{E} \cup \{(x + 0)[x := a] = x[x := a]\}$.
These follow from $0 + 0 =_{\mathcal{E}} 0$ and $s(a) + 0 =_{\mathcal{E}} s(a + 0) =_{\mathcal{E}'} s(a)$, which proves $x + 0 =_{\mathcal{E}} x$ by Theorem 2.

Theorem 2 has the requirement that for all $t \in \mathcal{T}(\Sigma)_s$ there exists $u \in \mathcal{T}(C)_s$ such that $u \sim_{\mathcal{E}} t$. In Example 1 we have argued that this is the case in an informal manner. However, we would like to prove that this is the case formally. Theorem 5 will be helpful in many cases. This Theorem allows us to directly conclude that for all $t \in \mathcal{T}(\Sigma)_s$ there exists $u \in \mathcal{T}(C)_s$ such that $u \sim_{\mathcal{E}} t$, if Σ, C, and $\mathcal{E}$ satisfy certain requirements. First we will prove Lemma 1 which will be used in the proof of Theorem 5.

This lemma describes that all terms in $\mathcal{T}(\Sigma)$ but not in $\mathcal{T}(C)$ must be of a certain shape. More precisely, any term $t \in \mathcal{T}(\Sigma)$ such that $t \notin \mathcal{T}(C)$ must be of the shape $\mathbf{C}[f(u_1, \ldots, u_n)$ for a context $\mathbf{C}$, $f \in \Sigma \setminus C$ and $u_1, \ldots, u_n \in \mathcal{T}(C)$.

Lemma 1 (Shape of terms in $\mathcal{T}(\Sigma)$ but not in $\mathcal{T}(C)$)
Let $C \subseteq \Sigma$ and let $t \in \mathcal{T}(\Sigma)$, but $t \notin \mathcal{T}(C)$.
Then t is of the shape $\boldsymbol{C}[f(u_1, \ldots, u_n)]$ for $f \in \Sigma \setminus C$, $u_1, \ldots, u_n \in \mathcal{T}(C)$ and some context $\boldsymbol{C}$.

Proof. Since $t \in \mathcal{T}(\Sigma)$ but $t \notin \mathcal{T}(C)$, there has to be at least one symbol $f \in \Sigma \setminus C$ in t. Therefore we can describe t as $\mathbf{C}[f(w_1, \ldots, w_n)]$ for $w_i \in \mathcal{T}(\Sigma)$ for $i = 1, \ldots, n$. If $w_i \in \mathcal{T}(C)$ for $i = 1, \ldots, n$ then we are done. Otherwise there is at least one $w_i \in \mathcal{T}(\Sigma)$ but not in $\mathcal{T}(C)$.

We can now do a proof by induction on the size of t. Take one of the w_i that contains a symbol in $\Sigma \setminus C$. Since w_i is a subterm of t, the size of w_i is smaller than the size of t. Therefore we can write w_i as $\mathbf{D}[g(v_1, \ldots, v_m)]$ for a context $\mathbf{D}$, a function $g \in \Sigma \setminus C$, and terms $v_1, \ldots, v_m \in \mathcal{T}(C)$ by applying the induction hypothesis.

Then we can create a new context **C'** defined as $\mathbf{C}[f(w_1, \ldots, \mathbf{D}[*], \ldots, w_n)]$ where w_i is replaced by $\mathbf{D}[*]$. Now t is of the shape $\mathbf{C'}[g(v_1, \ldots, v_m)]$ for $v_1, \ldots, v_m \in \mathcal{T}(C)$ and $g \in \Sigma \setminus C$ so we have reached our goal. □

An important consequence of a rewrite system being strongly normalizing is that there must exist at least one normal form. This is described by the following Theorem.

Theorem 4 (Strongly normalizing normal form)
If a TRS R over Σ is strongly normalizing (terminating), then every term $t \in \mathcal{T}(\Sigma, \mathcal{X})$ has at least one normal form with respect to R.

Proof. Take an arbitrary term $t \in \mathcal{T}(\Sigma, \mathcal{X})$. If t is already a normal form, then we have reached our goal. Otherwise, we keep rewriting t as long as possible using R. Since R is terminating, after some number of steps n we cannot rewrite any further. This yields a rewriting sequence $t \to_R t_1 \to_R \ldots \to_R t_n$. Therefore t_n is a normal form of t. □

Let C be a subset of a set of function symbols Σ. Using Lemma 1 and Theorem 4 we can now prove that for all $t \in \mathcal{T}(\Sigma)$ there exists $v \in \mathcal{T}(C)$ such that $t \sim_{\mathcal{E}} v$, if the set of equations $\mathcal{E}$ over Σ is designed such that:

- The rewrite system R created by transforming each equation $l = r \in \mathcal{E}$ to a rewrite rule $l \to r$ is strongly normalizing.
- For all terms created from functions in $\Sigma \setminus C$ there is a rule that can be applied.
- If a rule $l \to r$ has $l \in \mathcal{T}(C, \mathcal{X})$ then also $r \in \mathcal{T}(C, \mathcal{X})$.

We can actually prove not only that there exists such a term v, we can prove that this term is a normal form of t according to R. All of this is described formally by the following Theorem.

Theorem 5
Let $\mathcal{E}$ be a set of equations over Σ and let C be a subset of Σ. Let R be the rewrite system obtained by converting each equation $l = r \in \mathcal{E}$ to a rewrite rule $l \to r$. Assume that R is strongly normalizing. Assume that for all $f \in \Sigma \setminus C$ and for all $u_1, \ldots, u_n \in \mathcal{T}(C)$ there exists a rule $l \to r \in R$, and a substitution $\sigma : \mathcal{X} \to \mathcal{T}(\Sigma)$ such that $l\sigma = f(u_1, \ldots, u_n)$.
Assume that for all $l \to r \in R$, if $l \in \mathcal{T}(C, \mathcal{X})$ then $r \in \mathcal{T}(C, \mathcal{X})$.
Then for all $t \in \mathcal{T}(\Sigma)$ there exists $v \in \mathcal{T}(C)$ such that $t \sim_{\mathcal{E}} v$ and v is an R-normal form of t.

Proof. Take an arbitrary $t \in \mathcal{T}(\Sigma)$. If $t \in \mathcal{T}(C)$ and t is a normal form then we are done.

If $t \in \mathcal{T}(C)$ and t is not a normal form then by the fact that R is strongly normalizing and as a consequence of the assumption that for all $l \to r \in R$, if $l \in \mathcal{T}(C, \mathcal{X})$ then $r \in \mathcal{T}(C, \mathcal{X})$ there must be a normal form $u \in \mathcal{T}(C)$ such that $t \to_{R^*} u$ and therefore also $t \sim_{\mathcal{E}} u$ (see Theorem 4).

If t is not in $\mathcal{T}(C)$ then by Lemma 1 t must be of the shape $\mathbf{C}[f(u_1, \ldots, u_n)]$ for $u_1, \ldots, u_n \in \mathcal{T}(C)$, a context $\mathbf{C}$ and a symbol $f \in \Sigma \setminus C$. We will prove this case by induction on $\to_R$. By the assumption there exists a rule $l \to r$ and a substitution σ such that $\mathbf{C}[f(u_1, \ldots, u_n)] = \mathbf{C}[l\sigma]$. Therefore we can rewrite as follows: $\mathbf{C}[f(u_1, \ldots, u_n)] = \mathbf{C}[l\sigma] \to_R \mathbf{C}[r\sigma]$ and by the induction hypothesis $\mathbf{C}[r\sigma] \sim_{\mathcal{E}} u$ where $u \in \mathcal{T}(C)$ is a normal form. So we now have $t = \mathbf{C}[f(u_1, \ldots, u_n)] = \mathbf{C}[l\sigma] \sim_{\mathcal{E}} \mathbf{C}[r\sigma] \sim_{\mathcal{E}} u$. □

Consider Example 1 again. Earlier we simply assumed that for all $t \in \mathcal{T}(\Sigma)$ there exists $v \in \mathcal{T}(C)$ such that $t \sim_{\mathcal{E}} v$. We will now show how to prove this by applying Theorem 5. After converting the set of equations to a rewrite system, we obtain $R = \{0{+}x \to x,\ s(x){+}y \to s(x{+}y)\}$. Choose monotonic functions $[0] = 1,\ [s](x) = x + 1,\ [+](x, y) = 2x + y$. Then R is strongly normalizing according to Theorem 1.

The second requirement says that for all $f \in \Sigma \setminus C$ and all $u_1, \ldots, u_n \in \mathcal{T}(C)$ there exists a rule $l \to r$ and a substitution $\sigma : \mathcal{X} \to \mathcal{T}(C)$ such that $l(\sigma) = f(u_1, \ldots, u_n)$. This clearly holds if we observe the equations for $+$: for any term $+(u_1, u_2)$ where $u_1, u_2 \in \mathcal{T}(C)$ there is always a rule that can be applied.

The third requirement holds vacuously since there are no rules with $l \in \mathcal{T}(C, \mathcal{X})$. Now we may conclude that for all $t \in \mathcal{T}(\Sigma)$ there exists $v \in \mathcal{T}(C)$ such that $t \sim_{\mathcal{E}} v$ by applying Theorem 5.

Although in this thesis we will mainly focus on applying this induction principle to the natural numbers, applications of this induction principle are not limited to natural numbers. It can be applied on arbitrary sets of equations and symbols. This is illustrated by the following example on lists constructed from the empty list defined by the constant nil of sort *list* and the operator $::$ with sorts $d, list \to list$ which adds an element to the beginning of a list. A function $\mathsf{append} : list, list \to list$ is defined which appends a list to another list. The goal is to prove that appending a list xs to the empty list nil is equal to the list xs itself.

Example 2 (List append nil)
Let the set of equations $\mathcal{E}$ be defined as follows: $\mathcal{E} = \{\mathsf{append}(nil, xs) = xs,\ \mathsf{append}(x :: xs, ys) = x :: \mathsf{append}(xs, ys)\}$ over $\Sigma = \{nil, ::, \mathsf{append}\}$. Define $C = \{nil, ::\}$.

Assume that for all $t \in \mathcal{T}(\Sigma)$ there exists $u \in \mathcal{T}(C)$ such that $u =_{\mathcal{E}} t$ (which can be proven by Theorem 5). The goal is to prove $\mathsf{append}(xs, nil) = xs$. In order to apply Theorem 2 we have to show $\mathsf{append}(xs, nil)[xs := nil] =_{\mathcal{E}} xs[xs := nil]$ and
$\mathsf{append}(xs, nil)[xs := a_1 :: a_2] =_{\mathcal{E}'} xs[xs := a_1 :: a_2]$ where $\mathcal{E}' = \mathcal{E} \cup \{\mathsf{append}(a_1, nil) = a_1, \mathsf{append}(a_2, nil) = a_2$.
The first requirement follows directly from $\mathsf{append}(nil, nil) =_{\mathcal{E}} nil$.
The second follows from $\mathsf{append}(a_1 :: a_2, nil) =_{\mathcal{E}} a_1 :: \mathsf{append}(a_2, nil) =_{\mathcal{E}'} a_1 :: a_2$.

Which proves $append(xs, nil) = xs$ by Theorem 2.

In some cases applying Theorem 2 can have interesting results. For example, consider the data type Boolean defined by the constants 0 and 1. Let C be defined as $C = \{0, 1\}$. If we apply Theorem 2 to this data type, we actually get a case distinction, where we have to prove that the equation holds if $x = 0$ and if $x = 1$. This is due to the fact that all functions in C are constants, hence there are no induction hypotheses and the application of the induction principle results in a simple case distinction. The following example shows this behavior:

Example 3 (Double negation)
Consider the data type Boolean defined by constructors $0 :\to b$ and $1 :\to b$. The function $\mathsf{not} : b \to b$ is the negation of a term of type b, and can be defined by the equations $\mathsf{not}(0) = 1$ and $\mathsf{not}(1) = 0$. Since 0 and 1 are the only ground terms of type b, we can prove that $\mathsf{not}(\mathsf{not}(x)) = x$. Let $C = \{0, 1\}$, now we have to prove that

$$\mathsf{not}(\mathsf{not}(x))[x := 0] =_{\mathcal{E}} x[x := 0],$$

and secondly

$$\mathsf{not}(\mathsf{not}(x))[x := 1] =_{\mathcal{E}} x[x := 1].$$

In this example there are no hypotheses. This is due to the fact that 0 *and* 1 *are both constructors with no inputs. Therefore no fresh constants are generated which also means there are no hypotheses, similarly to the case* 0 *for naturals. The proof is straightforward,* $\mathsf{not}(\mathsf{not}(0)) =_{\mathcal{E}} \mathsf{not}(1) =_{\mathcal{E}} 0$ *and* $\mathsf{not}(\mathsf{not}(1)) =_{\mathcal{E}} \mathsf{not}(0) =_{\mathcal{E}} 1$*, which proves* $\mathsf{not}(\mathsf{not}(x)) = x$ *by Theorem 2.*

3.1 Undecidability of ground convertibility

Unfortunately Theorem 2 is not complete. This means that it cannot be successfully applied in all cases. In fact, we can prove that convertibility for ground terms is *undecidable*, and therefore ground convertibility must also be undecidable, since $\sim_{\mathcal{E}}$ and $=_{\mathcal{E}}$ are equivalent for ground terms.

A decision problem is called undecidable if it can be proven that it is impossible to construct an algorithm that always leads to a correct answer. In this case the decision problem that we are dealing with is deciding whether two terms are ground convertible or not. We can prove undecidability of ground convergence by reducing it to a known undecidable problem P. This means that we first assume that our problem is decidable. If the problem is decidable, there must exist an algorithm that always outputs the correct answer. We then show that given this algorithm, we can construct a new algorithm that solves the undecidable problem P. The existence of this algorithm is a contradiction, which means that our initial assumption that the problem is decidable was false. One way to prove undecidability of ground convertibility is by reducing it to the *Post Correspondence Problem (PCP)*. Such a proof can be found in [23].

Theorem 6 (Undecidability of ground convertibility)
Ground convertibility is undecidable, meaning that there exists no algorithm which has as input two terms t, u *and a set of equations* $\mathcal{E}$ *and decides whether* $t =_{\mathcal{E}} u$ *holds.*

The following is an example where Theorem 2 fails to prove $f(x) =_{\mathcal{E}} 0$, although it clearly must be true if we observe the equations.

Example 4 (incompleteness example half)
Let $\mathcal{E}$ *be defined by the following equations:*

$$\mathcal{E} = \{h(0) = 0, h(s(0)) = 0, h(s(s(x))) = s(h(x)), f(0) = 0, f(s(x)) = f(h(x))\}.$$

Then $h(n)$ *represents* $\lfloor \frac{n}{2} \rfloor$*. Now it is possible to prove that* $f(x) =_{\mathcal{E}} 0$*. However, Theorem 2 is not complete and cannot be used to prove this example. We have* $\Sigma = \{0, s, h, f\}$ *and if we take* $C = \{0, s\}$ *we have that for every* $t \in \mathcal{T}(\Sigma)$ *there exists* $u \in \mathcal{T}(C)$ *such that* $u \sim_{\mathcal{E}} t$*. So it has to be proven that* $t[x := 0] =_{\mathcal{E}} u[x := 0]$ *and* $t[x := s(a)] =_{\mathcal{E}'} u[x := s(a)]$*, where* $\mathcal{E}' = \mathcal{E} \cup \{f(a) = 0\}$*. The first case easily follows from* $f(0) \sim_{\mathcal{E}} 0$*. However in the second case we get stuck: we cannot show* $f(s(a)) =_{\mathcal{E}'} 0$ *directly. We can try* $f(h(a)) =_{\mathcal{E}'} 0$ *since* $f(s(a)) \sim_{\mathcal{E}} f(h(a))$*, but this does not lead us anywhere either.*

This shows that directly applying Theorem 2 does not allow us to prove this example. This does not necessarily mean that Theorem 2 cannot be used to prove this equation at all, since introducing additional lemmas could be useful. Although we think this is unlikely to work, it is not clear how to prove that proving the equation using Theorem 2 remains impossible after adding lemmas.

3.2 Strong induction

Nevertheless we observe that in order to prove this example, a kind of strong induction would be helpful. Intuitively we know by observing the equations that $h(a)$ is (roughly) half of $s(a)$. Therefore, instead of the hypothesis that $f(a) =_{\mathcal{E}} 0$ holds, we need the stronger hypothesis that $f(x) =_{\mathcal{E}} 0$ holds for all x smaller than $s(a)$. Since $h(a)$ corresponds to one of these cases, if we can prove that $h(a)$ is smaller than $s(a)$, the induction hypothesis allows us to conclude that

$f(s(a)) =_{\mathcal{E}} f(h(a)) =_{\mathcal{E}'} 0$. One of the main goals of this thesis is to introduce a new induction variant that can prove these kinds of examples. This new kind of induction is described by Theorem 7 for the general case. We give the version for natural numbers separately (see Theorem 8), as we will focus on the natural numbers in our examples.

We would like to add the hypotheses that the goal holds for all terms smaller than $f(a_1, \ldots, a_n)$. Hence, we will first of all need some order on the terms that can be constructed. For any set of terms there should be a term that is minimal according to this order, hence we need a well-founded order on the terms. This is problematic, if we want to define such an relation on $\mathcal{T}(\Sigma)$ we are typically going to run into problems. Consider a term t in $\mathcal{T}(\Sigma)$ that is convertible with a term u in $\mathcal{T}(C)$. If we define the well-founded order $<$ on $\mathcal{T}(\Sigma)$, then either $t < u$ or $u < t$. However, since t and u are convertible we then have $t < t$ or $u < u$ which violates the requirement for a well-founded relation. If we define $<$ on $\mathcal{T}(C)$, a new problem arises. In the example we saw that $h(a)$ corresponds to a term in $\mathcal{T}(C)$ which is smaller than $s(a)$, hence we could use the fact that the equation holds for $h(a)$ as a hypothesis. However, if we define $<$ on $\mathcal{T}(C)$ this is not possible, since $h(a)$ is not in $\mathcal{T}(C)$ if we choose $C = \{0, s\}$.

Our solution to these problems is to define a function φ which maps every term in $\mathcal{T}(\Sigma)$ to a term in $\mathcal{T}(C)$. We already had the assumption that each term in $\mathcal{T}(\Sigma)$ is convertible to a term in $\mathcal{T}(C)$, hence such a function must exist. Now, rather than defining an order on $\mathcal{T}(\Sigma)$, we can define an order on $\mathcal{T}(C)$. However since $a_1, \ldots, a_n$ are not in the set $\mathcal{T}(C)$, we still cannot compare any term containing some a_i to a term in $\mathcal{T}(C)$. To solve this problem we substitute all a_i by terms in $\mathcal{T}(C)$. Thus a term v containing $a_1, \ldots, a_n$ is less than $f(w_1, \ldots, w_n)$ if and only if $v[a_1 := w_1, \ldots, a_n := w_n]$ is less than $f(w_1, \ldots, w_n)$ for any terms $w_1, \ldots, w_n \in \mathcal{T}(C)$ according to the well-founded order $<$.

This allows us to describe the additional hypotheses follows. Let v be a term of the same sort as f in $\mathcal{T}(\Sigma \cup \bigcup_{i=1}^{n} \{a_i\})$. We allow v to be used as a hypothesis if and only if substituting each a_i that occurs in v by all terms w_i in $\mathcal{T}(C)$ always yields a term v' such that $\varphi(v')$ is smaller than $f(w_1, \ldots, w_n)$.

Theorem 7
Let $\mathcal{E}$ be a set of equations over Σ. Let $C \subseteq \Sigma$ such that every $f : s_1, \ldots, s_n \to s \in C$ is of the same sort s and $s_i = s$ for all i. Assume that for all $t \in \mathcal{T}(\Sigma)_s$ there exists a term $u \in \mathcal{T}(C)$ such that $u \sim_{\mathcal{E}} t$. Assume φ is a function that maps every term in $\mathcal{T}(\Sigma)_s$ to such a term in $\mathcal{T}(C)$, so $\varphi : \mathcal{T}(\Sigma)_s \to \mathcal{T}(C)$ such that $\forall t \in \mathcal{T}(\Sigma)_s$ we have $t \sim_{\mathcal{E}} \varphi(t)$.

Let $t, u \in \mathcal{T}(\Sigma, \mathcal{X})$ be two terms of the same sort. Let $x \in \mathcal{X}_s$. For every $f : s_1, \ldots, s_n \to s \in C$

$$t[x := f(a_1, \ldots, a_n)] =_{\mathcal{E}'} u[x := f(a_1, \ldots, a_n)],$$

where $a_1, \ldots, a_n$ are fresh constants of sort s, and

$$\mathcal{E}' = \mathcal{E} \cup \{t[x := v] = u[x := v] \mid \varphi(v[a_1 := w_1, \ldots, a_n := w_n]) < f(w_1, \ldots, w_n)\},$$

for $v \in \mathcal{T}(\Sigma_s \cup \bigcup_{i=1}^{n} \{a_i\})$, for all $w_1, \ldots, w_n \in \mathcal{T}(C)$ and a well-founded binary relation $<$ on $\mathcal{T}(C)$.

Then $t =_{\mathcal{E}} u$.

Proof. Using the same arguments as in the proof of Theorem 2, we may restrict ourselves to the case where $\sigma(x) \in \mathcal{T}(C)$.

We will prove $t[x := t'] =_{\mathcal{E}} u[x := t']$ for $t' \in \mathcal{T}(C)$ by well-founded induction on t' using the well-founded binary relation $<$.

Then t' is of the shape $f(v_1, \ldots, v_n)$ for some symbol $f \in \mathcal{T}(C)$ of sort s and all $v_1, \ldots, v_n$ of sort s.

By the assumption it holds that $t[x := f(a_1, \ldots, a_n)] =_{\mathcal{E}'} u[x := f(a_1, \ldots, a_n)]$ for fresh constants $a_1, \ldots, a_n$ of sort s. Therefore we have by the definitions of $=_{\mathcal{E}}$ and $\sim_{\mathcal{E}}$

$$t[x := f(a_1, \ldots, a_n)]\sigma = t_1 \leftrightarrow_{\mathcal{E}'} \ldots \leftrightarrow_{\mathcal{E}'} t_n = u[x := f(a_1, \ldots, a_n)]\sigma.$$

Since $a_1, \ldots, a_n$ are all of sort s, and $v_1, \ldots, v_n$ are all of sort s, we can replace each a_i by a corresponding v_i. Let T_i denote $t_i[a_j := v_j]$ for all $j = 1, \ldots, n$.

If $t_i \leftrightarrow_{\mathcal{E}} t_{i+1}$ then $T_i \sim_{\mathcal{E}} T_{i+1}$ follows from the definition of T.

If $t_i \leftrightarrow_{\mathcal{E}' \setminus \mathcal{E}} t_{i+1}$ then there is some context C such that $t_i = C[t[x := b]]\tau$ and $t_{i+1} = C[u[x := b]]\tau$ or vice versa, for $b \in \mathcal{T}(\Sigma \cup \bigcup_{i=1}^{n}\{a_i\})$ such that $\varphi(b[a_1 := w_1, \ldots, a_n := w_n]) < f(w_1, \ldots, w_n)$ for all $w_1, \ldots, w_n \in \mathcal{T}(C)$. Then by the induction hypothesis $t[x := \varphi(b[a_1 := v_1, \ldots, a_n := v_n])]\sigma \sim_{\mathcal{E}} u[x := \varphi(b[a_1 := v_1, \ldots, a_n := v_n])]\sigma$. Therefore by the definition of φ we have $t[x := b[a_1 := v_1, \ldots, a_n := v_n]]\sigma \sim_{\mathcal{E}} u[x := b[a_1 := v_1, \ldots, a_n := v_n]]\sigma$ and therefore $T_i \sim_{\mathcal{E}} T_{i+1}$.

So in every case $T_i \sim_{\mathcal{E}} T_{i+1}$, and therefore $t[x := t']\sigma = T_1\sigma \sim_{\mathcal{E}} T_n\sigma = u[x := t']\sigma$. □

In this thesis the focus will be on applying Theorem 7 on the natural numbers, where C is defined as $C = \{0, s\}$. The following Theorem is a corollary of Theorem 7 and describes the application on natural numbers.

Theorem 8 (Strong induction variant natural numbers)
Let $\mathcal{E}$ be a set of equations over Σ. Let $C = \{0, s\} \subseteq \Sigma$. Assume (1) that for all $t \in \mathcal{T}(\Sigma)$ of sort natural there exists a term $u \in \mathcal{T}(C)$ such that $u \sim_{\mathcal{E}} t$. Assume φ is a function that maps every term in $\mathcal{T}(\Sigma)$ to such a term in $\mathcal{T}(C)$, so $\varphi : \mathcal{T}(\Sigma) \to \mathcal{T}(C)$ such that $\forall t \in \mathcal{T}(\Sigma)$ we have $t \sim_{\mathcal{E}} \varphi(t)$.

Let $t, u \in \mathcal{T}(\Sigma, \mathcal{X})$ be two terms of the same sort. Let $x \in \mathcal{X}$ be a variable of sort natural. Assume that (2)

$$t[x := 0] =_{\mathcal{E}} u[x := 0],$$

and (3)

$$t[x := s(a)] =_{\mathcal{E}'} u[x := s(a)],$$

where a is a fresh constant of sort natural and

$$\mathcal{E}' = \mathcal{E} \cup \{t[x := v] = u[x := v] \mid v \in \mathcal{T}(\Sigma \cup \{a\}) : \forall w \in \mathcal{T}(C) : \varphi(v[a := w]) < s(w)\},$$

for a well-founded binary relation $<$ on $\mathcal{T}(C)$ and all $w \in \mathcal{T}(C)$.

Then $t =_{\mathcal{E}} u$.

Since we intend to use the half function in multiple examples, we will now prove that if the standard less-than relation on natural numbers is used, $h(w) < s(w)$ for all $w \in \mathcal{T}(C)$. First we prove that $h(w)$ corresponds to $\lfloor \frac{w}{2} \rfloor$, that is $h(w) =_{\mathcal{E}} s^{\lfloor \frac{w}{2} \rfloor}(0)$ for all w. Here we use the notation $s^x(0)$ which corresponds to s applied x times to 0.

Lemma 2
Let $\mathcal{E}$ be a set of equations over Σ where $C = \{0, s\} \subseteq \Sigma$ and $h \in \Sigma$, such that:

$$\{h(0) = 0, h(s(0)) = 0, h(s(s(x))) = s(h(x))\} \subseteq \mathcal{E}.$$

Let R be the rewrite system obtain by converting all equations $l = r$ to rewrite rules $l \to r$. Let $\mathcal{E}$ be defined such that R is strongly normalizing and there are no critical pairs.

Assume that for all $t \in \mathcal{T}(C)$, t is a normal form.

Then $h(s^n(0))$ has a unique normal form $s^{\lfloor \frac{n}{2} \rfloor}(0)$ for all $n \in \mathbb{N}$.

Proof. We will prove that $h(s^n(0))$ has a unique normal form $s^{\lfloor \frac{n}{2} \rfloor}(0)$ by induction on n.

If $n = 0$ then $h(s^0(0)) = h(0) =_{\mathcal{E}} 0 = s^0(0) = s^{\lfloor \frac{0}{2} \rfloor}(0)$.

If $n = 1$ then $h(s^1(0)) = h(s(0)) =_{\mathcal{E}} 0 = s^0(0) = s^{\lfloor \frac{1}{2} \rfloor}(0)$.

If $n = m + 2$ then $h(s^n(0)) = h(s^{m+2}(0)) =_{\mathcal{E}} s(h(s^m(0))) = s(s^{\lfloor \frac{m}{2} \rfloor}) = s^{\lfloor \frac{m+2}{2} \rfloor} = s^{\lfloor \frac{n}{2} \rfloor}(0)$. □

Theorem 9
Let $\mathcal{E}$ be a set of equations over Σ where $C = \{0, s\} \subseteq \Sigma$ and $h \in \Sigma$, such that:

$$\{h(0) = 0, h(s(0)) = 0, h(s(s(x))) = s(h(x))\} \subseteq \mathcal{E}.$$

Define $<$ as the well-founded binary relation $0 < s(0) < s(s(0)) < \ldots$ on $\mathcal{T}(C)$.

Let R be the rewrite system obtain by converting all equations $l = r$ to rewrite rules $l \to r$. Let $\mathcal{E}$ be defined such that R is strongly normalizing and there are no critical pairs.

Assume that for all $t \in \mathcal{T}(C)$, t is a normal form.

Then for all $w \in \mathcal{T}(C)$, $h(w)$ has a unique normal form u such that $u < s(w)$.

Proof. Since $w \in \mathcal{T}(C)$ where $C = \{0, s\}$ we can describe w as $h(s^n(0))$ for $n \geq 0$. By Lemma 2 we have that $(h(s^n(0)))$ has a unique normal form u: $s^{\lfloor \frac{n}{2} \rfloor}(0)$. Therefore we need to show $s^{\lfloor \frac{n}{2} \rfloor}(0) < s(s^n(0)) = s^{n+1}(0)$. Clearly $\lfloor \frac{n}{2} \rfloor < n + 1$ for all $n \geq 0$, and therefore we can conclude $u < s(w)$. □

We now have all the ingredients to prove the example we previously could not prove by Theorem 2.

Example 5 (Repeatedly taking $\lfloor \frac{n}{2} \rfloor$ of a natural number yields 0)
Consider the same set of equations $\mathcal{E}$ as defined in Example 4 where we want to prove $f(x) =_{\mathcal{E}} 0$:

$$\mathcal{E} = \{h(0) = 0, h(s(0)) = 0, h(s(s(x))) = s(h(x)), f(0) = 0, f(s(x)) = f(h(x))\}.$$

Let $<$ be the standard 'less-than' relation on natural numbers such that $0 < s(0) < s(s(0)) < \ldots$. Define $[0] = 1$, $[s](x) = x + 2$, $[h](x) = [f](x) = x + 1$, then the left hand side of every rule has a greater weight than the right hand side so $\mathcal{E} = R$ is strongly normalizing according to Theorem 1.

Define $C = \{0, s\}$. Then for all $f \in \Sigma \setminus C$ and all $t \in \mathcal{T}(C)$ there exists a rule $l \to r$ and a substitution $\sigma : \mathcal{X} \to \mathcal{T}(C)$ such that $l(\sigma) = f(t)$. This can be observed from the equations for f and h. Any term in $\mathcal{T}(C)$ is either 0 or the successor of another term in $\mathcal{T}(C)$. For f there is the rule $f(0) = 0$ for 0 and the rule $f(s(x)) = f(h(x))$ for the successor terms in $\mathcal{T}(C)$. For h there is the rule $h(0) = 0$ for 0 and the rule $h(s(0)) = 0$ for $s(0)$. For any other term the rule $h(s(s(x))) = s(h(x))$ applies.

Therefore by Theorem 5 for all $t \in \mathcal{T}(\Sigma)$ there exists a term $u \in \mathcal{T}(C)$ such that $u \sim_{\mathcal{E}} t$ which means we can apply Theorem 8 to solve this example. We now have to show that $f(x)[x := 0] =_{\mathcal{E}} 0[x := 0]$ and $f(x)[x := s(a)] =_{\mathcal{E}'} 0[x := s(a)]$ where $\mathcal{E}' = \mathcal{E} \cup \{f(x)[x := v] = 0[x := v] \mid v \in \mathcal{T}(\Sigma \cup \{a\}) : \forall w \in \mathcal{T}(C) : \varphi(v[a := w]) < s(w)\}$.

The first requirement follows directly from $f(0) \sim_{\mathcal{E}} 0$.

The second requirement follows from $f(s(a)) \sim_{\mathcal{E}} f(h(a)) \sim_{\mathcal{E}'} 0$. In the last step we use the fact that $\varphi(h(w)) < s(w)$ for all $w \in \mathcal{T}(C)$ according to Theorem 9, therefore we may use the hypothesis $f(x)[x := h(a)] = 0[x := h(a)]$. This proves $f(x) =_{\mathcal{E}} 0$ by Theorem 8.

In this example, the relation $<$ is straightforward because it is the standard less-than relation on numbers. However, there are examples where more complicated well-founded relations are required.

Example 6
Consider the following set of equations:

$$\mathcal{E} = \{f(0) = 0, f(s(x)) = f(g(x)), g(0) = s(s(0)), g(s(0)) = 0, g(s(s(0))) = s(0), \\ g(s(s(s(x)))) = s(s(s(x)))\}.$$

While at first sight this might not be obvious, but it is possible to prove that $f(x) = 0$ which can be observed by the following

$$f(0) =_{\mathcal{E}} 0$$
$$f(s(0)) =_{\mathcal{E}} f(g(0)) =_{\mathcal{E}} f(s(s(0)) =_{\mathcal{E}} f(g(s(0))) =_{\mathcal{E}} f(0) =_{\mathcal{E}} 0$$
$$f(s(s(0))) =_{\mathcal{E}} f(g(s(0))) =_{\mathcal{E}} f(0) =_{\mathcal{E}} 0$$
$$f(s(s(s(0)))) =_{\mathcal{E}} f(g(s(s(0)))) =_{\mathcal{E}} f(s(0)) =_{\mathcal{E}} 0$$
$$f(s(s(s(s(0))))) =_{\mathcal{E}} f(g(s(s(s(0))))) =_{\mathcal{E}} f(s(s(s(0)))) =_{\mathcal{E}} 0$$
$$\ldots$$

Using the tool AProVE [10] we verified that the TRS created by transforming the equations to rewrite rules is terminating. Hence we can apply Theorem 5 to conclude that for each $t \in \mathcal{T}(\Sigma)$ *there exists* $u \in \mathcal{T}(C)$ *such that* $u \sim_{\mathcal{E}} t$.

By Theorem 8 we can use the hypothesis $f(g(a)) = 0$ *if we can show that* $\varphi(g(w)) < s(w)$ *for all* $w \in \mathcal{T}(C)$. *However if we use the standard less-than relation on natural numbers this is not the case, as* $g(0) = s(s(0)) \not< s(0)$. *Instead, we can define* $<$ *as* $0 < s(s(0)) < s(0) < s(s(s(0))) < s(s(s(s(0)))) < \ldots$. *Now* $\varphi(g(w)) < s(w)$ *for all* $w \in \mathcal{T}(C)$ *does hold.*

We can prove that for all $w \in \mathcal{T}(C)$, $g(w)$ *has a unique normal form* u *such that* $u < s(w)$ *where* $<$ *is defined as described earlier. Since* $w \in \mathcal{T}(C)$, *it is of the shape* $s^n(0)$ *for* $n \geq 0$
If $n = 0$ *then we can use the rule* $g(0) \to s(s(0))$ *and* $s(s(0)) < s(0)$ *to conclude* $\varphi(g(0)) < s(0)$.
If $n = 1$ *then we can use the rule* $g(s(0)) \to 0$ *and* $0 < s(s(0))$ *to conclude* $\varphi(g(s(0)) < s(s(0))$.
If $n = 2$ *then we can use the rule* $g(s(s(0))) \to s(0)$ *and* $s(0) < s(s(s(0)))$ *to conclude* $\varphi(g(s(s(0))) < s(s(s(0)))$.
Otherwise, if $n = m{+}3$ *then using the rule* $g(s(s(s(x)))) \to s(s(s(x)))$ *we have* $g(s(s(s(s^m(0))))) \to s(s(s(s^m(0))))$ *and* $s(s(s(s^m(0)))) < s(s(s(s(s^m(0)))))$.

Since each $g(w)$ *with* $w \in \mathcal{T}(C)$ *has a unique normal form* u, $\varphi(g(w))$ *corresponds to this normal form and we have shown that* $\varphi(g(w)) < s(w)$.

Hence we can apply Theorem 8. We may use the hypothesis $f(g(a)) = 0$ *to obtain* $f(s(a)) = f(g(a)) = 0$. *This allows us to prove that* $f(x) = 0$ *using Theorem 8.*

3.3 Double induction

Sometimes modifications of Theorem 2 are useful. For example a variant called *double induction* will be used in some of the examples involving natural numbers in this thesis. This is also described in [23]. The intuition behind double induction is that we apply Theorem 2 twice. This means that if we want to prove $t =_{\mathcal{E}} u$, we first prove the case $t[x := 0] =_{\mathcal{E}} u[x := 0]$ as usual and we want to prove the case for $t[x := s(a)] =_{\mathcal{E}} u[x := s(a)]$ using the hypothesis $t[x := a] =_{\mathcal{E}} u[x := a]$. However rather than proving this directly (which might not be possible), we apply induction again to prove this goal. This means that we then have to prove $t[x := s(0)] =_{\mathcal{E}} u[x := s(0)]$, and $t[x := s(s(a))] =_{\mathcal{E}} u[x := s(s(a))]$ where we can now use both the hypotheses $t[x := a] =_{\mathcal{E}} u[x := a]$ and $t[x := s(a)] =_{\mathcal{E}} u[x := s(a)]$ as a consequence of the first and second application of induction respectively.

Theorem 10 (Double induction)
Let $\mathcal{E}$ *be a set of equations over* Σ. *Let* $C = \{0, s\} \subseteq \Sigma$. *Assume that for all* $t \in \mathcal{T}(\Sigma)_{nat}$ *there exists* $u \in \mathcal{T}(C)_{nat}$ *such that* $u \sim_{\mathcal{E}} t$.

Let $t, u \in \mathcal{T}(\Sigma, \mathcal{X})$ *be two terms of the same sort. Let* x *be a variable of sort natural.*

Assume that

$$t[x := 0] =_{\mathcal{E}} u[x := 0],$$

$$t[x := s(0)] =_{\mathcal{E}} u[x := s(0)],$$

and

$$t[x := s(s(a)) =_{\mathcal{E}'} u[x := s(s(a))],$$

where a *is a fresh constant of sort natural and*

$$\mathcal{E}' = \mathcal{E} \cup \{t[x := a] = u[x := a], t[x := s(a)] = u[x := s(a)]\}.$$

Then $t =_{\mathcal{E}} u$.

Proof. Using the same arguments as in the proof of Theorem 2, we may restrict ourselves to the case where $\sigma(x) \in \mathcal{T}(C)$.

We will prove $t[x := v] =_{\mathcal{E}} u[x := v]$ for $v \in \mathcal{T}(C)$ by induction on the size of v. Since we defined C as $C = \{0, s\}$ there are two cases: $v = 0$ or $v = s(v')$ for some $v' \in \mathcal{T}(C)$.

If $v = 0$ then we need to show $t[x := 0] =_{\mathcal{E}} u[x := 0]$, which holds by the assumptions.
If $v = s(v')$ then there are two cases for v': $v' = 0$ or $v' = s(v'')$ for some $v'' \in \mathcal{T}(C)$.
For the first case where $v' = 0$, we have $v = s(v') = s(0)$.
Then we need to show $t[x := s(0)] =_{\mathcal{E}} u[x := s(0)]$, which follows from the assumptions.
For the second case we have that $v = s(v') = s(s(v''))$.
From the assumptions we know that

$$t[x := s(s(a))] =_{\mathcal{E}'} u[x := s(s(a))].$$

Since a is a fresh constant of sort natural and v'' is of sort natural, we can replace every a in this conversion by v'' to obtain

$$t[x := s(s(v''))] =_{\mathcal{E}''} u[x := s(s(v''))].$$

Therefore $t[x := s(s(v''))]\sigma \sim_{\mathcal{E}''} u[x := s(s(v''))]\sigma$ for all ground substitutions σ by the definition of $=_{\mathcal{E}''}$.

Since the size of v'' and $s(v'')$ is less than the size of v, we have that $t[x := v''] =_{\mathcal{E}} u[x := v'']$ and $t[x := s(v'')] =_{\mathcal{E}} u[x := s(v'')]$ by the induction hypothesis. Now we can apply Theorem 3 which yields $t =_{\mathcal{E}} u$. □

Chapter 4

Infinite sequences

The terms we considered in the equality problems so far were all finite terms. However, we can extend the techniques we used to infinite terms. A one-sided infinite sequence can be seen as a mapping from natural numbers to some data type, which can for instance be the natural numbers or booleans. Infinite sequences are typically specified using either the : operator, or with the head and tail operators.

The : operator puts a data element element in front of an infinite sequence. The head operator simply extracts the first element of an infinite sequence, and tail extracts the remainder of the sequence (so everything that comes after the head). If we would like to describe the infinite sequence with 0 at every position, we can define it as $\mathsf{zeros} = 0 : \mathsf{zeros}$ using the : operator. Alternatively, we can specify the same sequence using head and tail and the following two equations: $\mathsf{head}(\mathsf{zeros}) = 0$ and $\mathsf{tail}(\mathsf{zeros}) = \mathsf{zeros}$. Throughout this paper we will generally specify infinite sequences using the head and tail operators, however in rare cases we may use the : operator.

When dealing with infinite sequences there are at least two sorts involved, a sort i representing the infinite sequences and a sort d representing the basic data type. Therefore the types head and tail are $\mathsf{head} : i \to d$ and $\mathsf{tail} : i \to i$. The notation $[t]$ describes the infinite sequence represented by the term t. If t is a ground term then $[t]$ is just the single infinite sequence represented by t. If t contains variables then $[t]$ is a function in these variables yielding the resulting infinite sequence [23].

The main technique to automatically prove equality of infinite objects is *Circular Coinduction* [20]. This provides a technique to deal with infinite data, and prove equality of infinite data objects like infinite sequences. This is done by combining equational reasoning on finite terms with the circular coinduction principle. We will first describe this approach and then describe a different approach that allows us to reuse the induction methods from Chapter 3. Both approaches are closely related and based on the idea that to prove two infinite sequences are equal, we show that every element of first sequence is equal to the element at the same position of the second sequence. Hence, if t, u are two infinite sequences, we prove $[t] = [u]$ by proving $[t](k) = [u](k)$ for all natural numbers k.

Throughout this chapter there are some functions that will commonly be used. Rather than giving the definitions of these functions in every example, we have gathered them in Table 4.1. In examples from now on we will in general only explicitly give the equations for operators or lemmas that are not included in this table, unless operators are defined by alternative equations.

Function	**Equations**	
$\mathsf{take}(x, xs)$	$\mathsf{take}(0, xs) = \mathsf{head}(xs)$	$\mathsf{take}(s(x), xs) = \mathsf{take}(x, \mathsf{tail}(xs))$
$\mathsf{zip}(xs, ys)$	$\mathsf{head}(\mathsf{zip}(xs, ys)) = \mathsf{head}(xs)$	$\mathsf{tail}(\mathsf{zip}(xs, ys)) = \mathsf{zip}(ys, \mathsf{tail}(xs))$
ones	$\mathsf{head}(\mathsf{ones}) = 1$	$\mathsf{tail}(\mathsf{ones}) = \mathsf{ones}$
zeros	$\mathsf{head}(\mathsf{zeros}) = 0$	$\mathsf{tail}(\mathsf{zeros}) = \mathsf{zeros}$
$\mathsf{inv}(xs)$	$\mathsf{head}(\mathsf{inv}(xs)) = \mathsf{not}(\mathsf{head}(xs))$	$\mathsf{tail}(\mathsf{inv}(xs)) = \mathsf{inv}(\mathsf{tail}(xs))$
$\mathsf{not}(b)$	$\mathsf{not}(0) = 1$	$\mathsf{not}(1) = 0$
alt	$\mathsf{head}(\mathsf{alt}) = 0$ $\mathsf{head}(\mathsf{tail}(\mathsf{alt})) = 1$	$\mathsf{tail}(\mathsf{tail}(\mathsf{alt})) = \mathsf{alt}$
$h(x)$	$h(0) = 0$ $h(s(0)) = 0$	$h(s(s(x))) = s(h(x))$
$d : xs$	$\mathsf{head}(d : xs) = d$	$\mathsf{tail}(d : xs) = xs$
$\mathsf{even}(xs)$	$\mathsf{head}(\mathsf{even}(xs)) = \mathsf{head}(xs)$	$\mathsf{tail}(\mathsf{even}(xs)) = \mathsf{even}(\mathsf{tail}(\mathsf{tail}(xs)))$
$+(x, y)$	$+(0, y) = y$	$+(s(x), y) = s(+(x, y))$
$S(xs)$	$\mathsf{head}(S(xs)) = s(\mathsf{head}(xs))$	$\mathsf{tail}(S(xs)) = S(\mathsf{tail}(xs))$
$\mathsf{plus}(xs, ys)$	$\mathsf{head}(\mathsf{plus}(xs, ys)) = +(\mathsf{head}(xs), \mathsf{head}(ys))$	$\mathsf{tail}(\mathsf{plus}(xs, ys)) = \mathsf{plus}(\mathsf{tail}(xs), \mathsf{tail}(ys))$

Table 4.1: Commonly used functions and their equations

4.1 Circular Coinduction

Circular coinduction systematically searches for periodic behaviors of the two infinite terms to prove equivalent. It derives the behavioral equational task until one obtains on every path either a truth or a cycle. Since each path ends up in a cycle it cannot be shown that the two terms are different, so they must be behaviorally equivalent [20, 17].

In circular coinduction equality of two infinite sequences l and r is proved by showing that $\mathsf{head}(l) = \mathsf{head}(r)$ by rewriting using the equations in $\mathcal{E}$ and $\mathsf{fr}(\mathsf{tail}(l)) = \mathsf{fr}(\mathsf{tail}(r))$ by rewriting using the equations in $\mathcal{E}$ and additionally the hypothesis $\mathsf{fr}(l) = \mathsf{fr}(r)$. Without freezing, the last rule would vacuously hold because $\mathsf{tail}(l) = \mathsf{tail}(r)$ if we have the hypothesis $l = r$. Freezing prevents this unwanted behavior and makes it possible to only allow the coinduction hypothesis to be used on the top level, and not on subterms. Freezing can easily be achieved by extending the signature Σ with a new sort *Frozen* and the operation $\mathsf{fr} : s \to \mathit{Frozen}$ for each sort s. Double freezing is not allowed, so s refers only to the sorts that were originally in Σ. Using our notation we can formulate the basic circular coinduction principle as follows for one-sided infinite sequences, based on the definition given in [24]:

Theorem 11 (Circular Coinduction)
Let $\mathcal{E}$ be a set of equations over Σ not containing the symbol fr *and the sort Frozen. For each sort s that occurs in Σ, add the symbol* $\mathsf{fr} : s \to \mathit{Frozen}$.
Assume that

$$\mathsf{head}(t) =_{\mathcal{E}} \mathsf{head}(u) \text{ and } \mathsf{fr}(\mathsf{tail}(t)) =_{\mathcal{E}'} \mathsf{fr}(\mathsf{tail}(u))$$

for $\mathcal{E}' = \mathcal{E} \cup \{\mathsf{fr}(t) = \mathsf{fr}(u)\}$.
Then $[t] = [u]$.

To show how this can be applied we will consider an example. We will now prove that $\mathsf{zip}(\mathsf{even}(xs), \mathsf{odd}(xs)) = xs$. Here $\mathsf{even}(xs)$ defines the sequence consisting of all elements of xs at even positions $(0, 2, 4, \ldots)$, and $\mathsf{odd}(xs)$ defines the sequence consisting of all elements of xs at odd positions $(1, 3, 5, \ldots)$. The operator $\mathsf{zip}(xs, ys)$ defines the sequence which is created by alternatingly taking elements from xs and ys. Therefore taking zip of the even and odd elements of some sequence xs should yield the original sequence xs.

Example 7
Let $\mathcal{E}$ consist of the equations for zip*, and the equations for the operators* odd *and* even *which are defined as follows:*

$$\mathsf{head}(\mathsf{even}(xs)) = \mathsf{head}(xs) \quad \mathsf{tail}(\mathsf{even}(xs)) = \mathsf{odd}(\mathsf{tail}(xs))$$
$$\mathsf{odd}(xs) = \mathsf{even}(\mathsf{tail}(xs))$$

We can now prove $\mathsf{zip}(\mathsf{even}(xs), \mathsf{odd}(xs)) = xs$ *as follows using the circular coinduction method (Theorem 11). The first requirement is to show that* $\mathsf{head}(\mathit{zip}(\mathsf{even}(xs), \mathsf{odd}(xs))) = \mathsf{head}(xs)$. *The second requirement is to show that* $\mathsf{fr}(\mathsf{tail}(\mathsf{zip}(\mathsf{even}(xs), \mathsf{odd}(xs)))) = \mathsf{fr}(\mathsf{tail}(xs))$ *where we are allowed to use the hypothesis* $\mathsf{fr}(\mathsf{zip}(\mathsf{even}(xs), \mathsf{odd}(xs))) = \mathsf{fr}(xs)$ *(IH). This leads to the following conversions.*

$$\begin{array}{lll}
\mathsf{head}(\mathsf{zip}(\mathsf{even}(xs), \mathsf{odd}(xs))) & =_{\mathcal{E}} \; \mathsf{head}(\mathsf{even}(xs)) & \mathsf{zip} \\
& =_{\mathcal{E}} \; \mathsf{head}(xs) & \mathsf{even} \\
\\
\mathsf{fr}(\mathsf{tail}(\mathsf{zip}(\mathsf{even}(xs), \mathsf{odd}(xs)))) & =_{\mathcal{E}} \; \mathsf{fr}(\mathsf{zip}(\mathsf{odd}(xs), \mathsf{tail}(\mathsf{even}(xs)))) & \mathsf{zip} \\
& =_{\mathcal{E}} \; \mathsf{fr}(\mathsf{zip}(\mathsf{odd}(xs), \mathsf{odd}(\mathsf{tail}(xs)))) & \mathsf{even} \\
& =_{\mathcal{E}} \; \mathsf{fr}(\mathsf{zip}(\mathsf{even}(\mathsf{tail}(xs)), \mathsf{odd}(\mathsf{tail}(xs)))) & \mathsf{odd} \\
& =_{\mathcal{E}'} \; \mathsf{fr}(\mathsf{tail}(xs)) & \mathit{IH}
\end{array}$$

In this conversion the rightmost column indicates the equation that was used to rewrite the previous term to the new term.
These two conversions show that both requirements for circular coinduction hold, therefore by Theorem 11 we may conclude $[\mathsf{zip}(\mathsf{even}(xs), \mathsf{odd}(xs))] = [xs]$.

4.2 Induction

In this paper we use a different approach than circular coinduction. This is based on the techniques described in [23]. Since infinite sequences can be considered as a mapping from natural numbers to a data type, let $[t](x)$ describe the element at position x of the infinite sequence t. Now an alternative approach is to prove that two infinite sequences t and u are equal by showing that at each position x, the element at position x of t and u are equal. More specifically, we show that $[t](x) = [u](x)$ for all natural numbers x. The key idea is that we can define an operator take, which takes an element at a specific position of a sequence.

We define the operator $\mathsf{take} : nat, i \to d$ such that $\mathsf{take}(k, t)$ describes the kth element of the sequence $[t]$. We can define this operator using the following two equations:

$$\mathsf{take}(0, xs) = \mathsf{head}(xs)$$
$$\mathsf{take}(s(x), xs) = \mathsf{take}(x, \mathsf{tail}(xs))$$

Where the first equation describes that the first element (at index 0) of the sequence corresponds to the head of the sequence. The second equation describes that the $k + 1$-th element of the sequence corresponds to the k-th element of the tail of the sequence. This fully describes the take operator for one-sided infinite sequences, which can be seen as a mapping from natural numbers to a data type, since every natural number is either of the shape 0, or $s(x)$.

We described earlier that two infinite sequences are considered equal if the elements at every position are the same. We can now define equality of two infinite sequences as follows using the take operator.

Theorem 12 (Equality of infinite sequences using take)
Let the set $\mathcal{E}$ of equations contain the two equations $\mathsf{take}(0, xs) = \mathsf{head}(xs)$ and $\mathsf{take}(s(x), xs) = \mathsf{take}(x, \mathsf{tail}(xs))$. Let t,u be terms of sort i satisfying $\mathsf{take}(x, t) =_{\mathcal{E}} \mathsf{take}(x, u)$.
Then $[t] = [u]$.

Proof. Equality of infinite sequences is defined such that two infinite sequences t and u are equal if and only if $[t](x) = [u](x)$ for all natural numbers x. Assume towards a contradiction that $\mathsf{take}(x, t) =_{\mathcal{E}} \mathsf{take}(x, u)$ is satisfied but $[t] \neq [u]$. Then there must be at least one position k such that the elements at position k are different for the two sequences, i.e. $[t](k) \neq [u](k)$. However if we represent k as a natural number using 0 and s, then we have $\mathsf{take}(k, t) =_{\mathcal{E}} \mathsf{take}(k, u)$ which contradicts the fact that the elements at position k are different. Hence, the initial assumption that $[t] \neq [u]$ must be false, which proves $[t] = [u]$. □

To prove equality of two sequences we can now reuse our induction methods from the previous chapter to do induction on x which will help us prove $\mathsf{take}(x, t) = \mathsf{take}(x, u)$ and therefore as a consequence of Theorem 12 also $[t] = [u]$.

4.2.1 Basic induction

Combining this definition of equality with the main induction Theorem 2 in the previous chapter yields the following Theorem for proving equality of infinite sequences. We will show that this is equivalent to Circular Coinduction described in the previous section.

Theorem 13 (Basic induction infinite sequences)
Let the set $\mathcal{E}$ of equations contain the two equations $\mathsf{take}(0, xs) = \mathsf{head}(xs)$ and $\mathsf{take}(s(x), xs) = \mathsf{take}(x, \mathsf{tail}(xs))$. Let t,u be terms of sort i satisfying $\mathsf{take}(0, t) =_{\mathcal{E}} \mathsf{take}(0, u)$ and $\mathsf{take}(s(a), t) =_{\mathcal{E}'} \mathsf{take}(s(a), u)$ for $\mathcal{E}' = \mathcal{E} \cup \{\mathsf{take}(a, t) = \mathsf{take}(a, u)\}$. Then $[t] = [u]$.

We can now prove the same example as before, using this Theorem rather than circular coinduction.

Example 8
Let $\mathcal{E}$ consist of the equations for $\mathsf{zip}, \mathsf{take}$ *and the equations for* odd *and* even *as given in Example 7.*

We can now prove $[\mathsf{zip}(\mathsf{even}(xs), \mathsf{odd}(xs))] = [xs]$ *as follows using Theorem 13. The first requirement is to show that* $\mathsf{take}(0, \mathsf{zip}(\mathsf{even}(xs), \mathsf{odd}(xs))) =_{\mathcal{E}} \mathsf{take}(0, xs)$. *The second requirement is to show* $\mathsf{take}(s(a), \mathsf{zip}(\mathsf{even}(xs), \mathsf{odd}(xs))) =_{\mathcal{E}'} \mathsf{take}(s(a), xs)$ *where* $\mathcal{E}' = \mathcal{E} \cup \{\mathsf{take}(a, \mathsf{zip}(\mathsf{even}(xs), \mathsf{odd}(xs))) = \mathsf{take}(a, xs)$ *(IH).*

$$
\begin{array}{lll}
\mathsf{take}(0, \mathsf{zip}(\mathsf{even}(xs), \mathsf{odd}(xs))) & =_{\mathcal{E}} \mathsf{head}(\mathsf{zip}(\mathsf{even}(xs), \mathsf{odd}(xs))) & \mathsf{take} \\
& =_{\mathcal{E}} \mathsf{head}(\mathsf{even}(xs)) & \mathsf{zip} \\
& =_{\mathcal{E}} \mathsf{head}(xs) & \mathsf{even} \\
& =_{\mathcal{E}} \mathsf{take}(0, xs) & \mathsf{take} \\
\\
\mathsf{take}(s(a), \mathsf{zip}(\mathsf{even}(xs), \mathsf{odd}(xs))) & =_{\mathcal{E}} \mathsf{take}(a, \mathsf{tail}(\mathsf{zip}(\mathsf{even}(xs), \mathsf{odd}(xs)))) & \mathsf{take} \\
& =_{\mathcal{E}} \mathsf{take}(a, \mathsf{zip}(\mathsf{odd}(xs), \mathsf{tail}(\mathsf{even}(xs)))) & \mathsf{zip} \\
& =_{\mathcal{E}} \mathsf{take}(a, \mathsf{zip}(\mathsf{odd}(xs), \mathsf{odd}(\mathsf{tail}(xs)))) & \mathsf{even} \\
& =_{\mathcal{E}} \mathsf{take}(a, \mathsf{zip}(\mathsf{even}(\mathsf{tail}(xs)), \mathsf{odd}(\mathsf{tail}(xs)))) & \mathsf{odd} \\
& =_{\mathcal{E}} \mathsf{take}(a, \mathsf{tail}(xs)) & \mathit{IH} \\
& =_{\mathcal{E}} \mathsf{take}(s(a), xs) & \mathsf{take}
\end{array}
$$

This proves the equation by Theorem 13.

Example 8 showed that we could also prove the example we gave for circular coinduction using Theorem 13. The following Theorem describes that this holds in general, i.e. we can prove $[l] = [r]$ using circular coinduction if and only if we can prove $[l] = [r]$ using Theorem 13.

Theorem 14 (Relation circular coinduction and Theorem 13)
Proofs of $[l] = [r]$ by the circular coinduction method formalized in Theorem 11 and proofs by Theorem 13 can be transformed to each other. Therefore we can prove $[l] = [r]$ using circular coinduction if and only if we can prove $[l] = [r]$ using Theorem 13.

Proof. First assume we have a proof of $[l] = [r]$ by circular coinduction as described in 11. Then the first requirement of this proof is $\mathsf{head}(l) =_{\mathcal{E}} \mathsf{head}(r)$. Then also the first requirement of Theorem 13 is satisfied:

$$\mathsf{take}(0, l) =_{\mathcal{E}''} \mathsf{head}(l) =_{\mathcal{E}} \mathsf{head}(r) =_{\mathcal{E}''} \mathsf{take}(0, r)$$

where $\mathcal{E}''$ is $\mathcal{E}$ extended with the equations for take as described in Theorem 13.

For the second requirement we have $\mathsf{fr}(\mathsf{tail}(l)) =_{\mathcal{E}'} \mathsf{fr}(\mathsf{tail}(r))$. We now replace every occurrence of $\mathsf{fr}(x)$ by $\mathsf{take}(a, x)$. This yields

$$\mathsf{take}(s(a), l) =_{\mathcal{E}'''} \mathsf{take}(a, \mathsf{tail}(l)) =_{\mathcal{E}'''} \mathsf{take}(a, \mathsf{tail}(r)) =_{\mathcal{E}'''} \mathsf{take}(s(a), r)$$

where $\mathcal{E}'''$ is $\mathcal{E}'$ extended with the equations for take as described in Theorem 13.

Together these two parts form a valid proof by Theorem 13.

Now assume that we have a proof by Theorem 13. Then the first part yields $\mathsf{take}(0, l) =_{\mathcal{E}} \mathsf{take}(0, r)$. Clearly this also yields $\mathsf{head}(l) =_{\mathcal{E}} \mathsf{head}(r)$, since $\mathsf{take}(0, l) =_{\mathcal{E}''} \mathsf{head}(l)$ and $\mathsf{take}(0, r) =_{\mathcal{E}''} \mathsf{head}(r)$. Combining these yields $\mathsf{head}(l) =_{\mathcal{E}''} \mathsf{take}(0, l) =_{\mathcal{E}} \mathsf{take}(0, r) =_{\mathcal{E}''} \mathsf{head}(r)$.

The second part gives us $\mathsf{take}(s(a), l) =_{\mathcal{E}'} \mathsf{take}(s(a), r)$ where $\mathcal{E}' = \mathcal{E} \cup \{\mathsf{take}(a, l) = \mathsf{take}(a, r)\}$. Replacing every instance of $\mathsf{take}(s^k(a), x)$ by $\mathsf{fr}(\mathsf{tail}^k(x))$ yields a new conversion for $\mathsf{fr}(\mathsf{tail}(l)) =_{\mathcal{E}'''} \mathsf{fr}(\mathsf{tail}(r))$ where $\mathcal{E}'''$ is $\mathcal{E}'$ with every instance of take replaced, as described above. These two parts together are exactly what we need for a proof using the circular coinduction technique. □

4.2.2 Double induction

Since the variable x in Definition 12 is of type nat, we can also combine this Definition with the double induction variant as described in Theorem 10. This yields the following double induction variant of Theorem 13.

Theorem 15 (Double induction infinite sequences)
Let the set $\mathcal{E}$ of equations contain the two equations $\mathsf{take}(0, xs) = \mathsf{head}(xs)$ *and* $\mathsf{take}(s(x), xs) = \mathsf{take}(x, \mathsf{tail}(xs))$. *Let t,u be terms of sort i satisfying* $\mathsf{take}(0, t) =_{\mathcal{E}} \mathsf{take}(0, u)$, $\mathsf{take}(s(0), t) =_{\mathcal{E}} \mathsf{take}(s(0), u)$ *and* $\mathsf{take}(s(s(a)), t) =_{\mathcal{E}'} \mathsf{take}(s(s(a)), u)$ *for* $\mathcal{E}' = \mathcal{E} \cup \{\mathsf{take}(a, t) = \mathsf{take}(a, u), \mathsf{take}(s(a), t) = \mathsf{take}(s(a), u)\}$. *Then* $[t] = [u]$.

In practice this variant of Theorem 13 turns out to be useful in many cases when certain operators like zip are used.

Example 9
We can now prove that if we zip an infinite sequence xs with itself: $\mathsf{zip}(xs, xs)$, *then the element at position x of the zipped sequence is equal to the element at position $h(x)$ of the original sequence. Here the double induction variant is used, as described in Theorem 15.*

The goal is to prove: $\mathsf{take}(x, \mathsf{zip}(y, y)) = \mathsf{take}(h(x), y)$

$\mathsf{take}(0, \mathsf{zip}(y, y))$	$=_{\mathcal{E}}$	$\mathsf{head}(\mathsf{zip}(y, y))$	take
	$=_{\mathcal{E}}$	$\mathsf{head}(y)$	zip
	$=_{\mathcal{E}}$	$\mathsf{take}(0, y)$	take
	$=_{\mathcal{E}}$	$\mathsf{take}(h(0), y)$	h
$\mathsf{take}(s(0), \mathsf{zip}(y, y))$	$=_{\mathcal{E}}$	$\mathsf{take}(0, \mathsf{tail}(\mathsf{zip}(y, y)))$	take
	$=_{\mathcal{E}}$	$\mathsf{head}(\mathsf{tail}(\mathsf{zip}(y, y)))$	take
	$=_{\mathcal{E}}$	$\mathsf{head}(\mathsf{zip}(y, \mathsf{tail}(y)))$	zip
	$=_{\mathcal{E}}$	$\mathsf{head}(y)$	zip
	$=_{\mathcal{E}}$	$\mathsf{take}(0, y)$	take
	$=_{\mathcal{E}}$	$\mathsf{take}(h(s(0)), y)$	h
$\mathsf{take}(s(s(a)), \mathsf{zip}(y, y))$	$=_{\mathcal{E}}$	$\mathsf{take}(s(a), \mathsf{tail}(\mathsf{zip}(y, y)))$	take
	$=_{\mathcal{E}}$	$\mathsf{take}(a, \mathsf{tail}(\mathsf{tail}(\mathsf{zip}(y, y))))$	take
	$=_{\mathcal{E}}$	$\mathsf{take}(a, \mathsf{tail}(\mathsf{zip}(y, \mathsf{tail}(y))))$	zip
	$=_{\mathcal{E}}$	$\mathsf{take}(a, \mathsf{zip}(\mathsf{tail}(y), \mathsf{tail}(y)))$	zip
	$=_{\mathcal{E}'}$	$\mathsf{take}(h(a), \mathsf{tail}(y))$	*IH:* $take(a, zip(y, y) = take(h(a), y)$
	$=_{\mathcal{E}}$	$\mathsf{take}(s(h(a)), y)$	take
	$=_{\mathcal{E}}$	$\mathsf{take}(h(s(s(a))), y)$	h

Which proves $\mathsf{take}(x, \mathsf{zip}(y, y)) = \mathsf{take}(h(x), y)$ *by Theorem 15.*

The following example is more complex than the ones we have seen so far. We can still prove this example using our techniques.

Example 10
Consider the infinite sequence $B = 0 : \mathsf{zip}(\mathsf{inv}(B), \mathsf{inv}(B))$. *It has the following shape:*

$$0 : 1 : 1 : 0 : 0 : 0 : 0 : 1^8 : 0^{16} : \ldots$$

We can observe that taking the elements at the positions $2^x - 1$ seems to yield the sequence alt, *defined as $0 : 1 : alt$:*
$2^0 - 1 = 0$, *element at position 0 is 0.*
$2^1 - 1 = 1$, *element at position 1 is 1.*
$2^2 - 1 = 3$, *element at position 3 is 0.*
$2^3 - 1 = 7$, *element at position 7 is 1.*
...
If this is indeed the case, we should be able to prove this automatically using our techniques.

We define the function f, such that $f(x)$ represents $2^x - 1$. This requires us to first define a function d, such that $d(x)$ represents $x \cdot 2$. These two functions can be described as follows:

$$d(0) = 0,\ d(s(x)) = s(s(d(x)))$$

$$f(0) = 0,\ f(s(x)) = s(d(f(x)))$$

We can now prove $\mathsf{take}(f(x), B) = \mathsf{take}(x, alt)$ *with some additional lemmas. The proofs of these lemmas can be found in Appendix B.*

Number	***Equation***
1	$h(d(x)) = x$
2	$\mathsf{take}(x, \mathsf{zip}(y, y)) = \mathsf{take}(h(x), y)$
3	$\mathsf{inv}(\mathsf{zip}(\mathsf{inv}(x), \mathsf{inv}(x))) = \mathsf{zip}(x, x)$

Goal: $\mathsf{take}(f(x), B) = \mathsf{take}(x, \mathsf{alt})$

$\mathsf{take}(f(0), B)$	$=_\mathcal{E}$	$\mathsf{take}(0), B)$	f
	$=_\mathcal{E}$	$\mathsf{head}(B)$	take
	$=_\mathcal{E}$	0	B
	$=_\mathcal{E}$	$\mathsf{head}(\mathsf{alt})$	alt
	$=_\mathcal{E}$	$\mathsf{take}(0, \mathsf{alt})$	take
$\mathsf{take}(f(s(0)), B)$	$=_\mathcal{E}$	$\mathsf{take}(s(d(f(0))), B)$	f
	$=_\mathcal{E}$	$\mathsf{take}(s(d(0)), B)$	f
	$=_\mathcal{E}$	$\mathsf{take}(s(0), B)$	d
	$=_\mathcal{E}$	$\mathsf{take}(0, \mathsf{tail}(B))$	take
	$=_\mathcal{E}$	$\mathsf{head}(\mathsf{tail}(B))$	take
	$=_\mathcal{E}$	$\mathsf{head}(\mathsf{zip}(\mathsf{inv}(B), \mathsf{inv}(B)))$	B
	$=_\mathcal{E}$	$\mathsf{head}(\mathsf{inv}(B))$	zip
	$=_\mathcal{E}$	$\mathsf{not}(\mathsf{head}(B))$	inv
	$=_\mathcal{E}$	$\mathsf{not}(0)$	B
	$=_\mathcal{E}$	1	not
	$=_\mathcal{E}$	$\mathsf{head}(\mathsf{tail}(\mathsf{alt}))$	alt
	$=_\mathcal{E}$	$\mathsf{take}(0, \mathsf{tail}(\mathsf{alt}))$	take
	$=_\mathcal{E}$	$\mathsf{take}(s(0), \mathsf{alt})$	take
$\mathsf{take}(f(s(s(a))), B)$	$=_\mathcal{E}$	$\mathsf{take}(s(d(f(s(a)))), B)$	f
	$=_\mathcal{E}$	$\mathsf{take}(s(d(s(d(f(a))))), B)$	f
	$=_\mathcal{E}$	$\mathsf{take}(d(s(d(f(a)))), \mathsf{tail}(B)$	take
	$=_\mathcal{E}$	$\mathsf{take}(s(s(d(d(f(a))))), \mathsf{tail}(B)$	d
	$=_\mathcal{E}$	$\mathsf{take}(s(d(d(f(a)))), \mathsf{tail}(\mathsf{tail}(B)))$	take
	$=_\mathcal{E}$	$\mathsf{take}(d(d(f(a))), \mathsf{tail}(\mathsf{tail}(\mathsf{tail}(B))))$	take
	$=_\mathcal{E}$	$\mathsf{take}(d(d(f(a))), \mathsf{tail}(\mathsf{tail}(\mathsf{zip}(\mathsf{inv}(B), \mathsf{inv}(B)))))$	B
	$=_\mathcal{E}$	$\mathsf{take}(d(d(f(a))), \mathsf{tail}(\mathsf{zip}(\mathsf{inv}(B), \mathsf{tail}(\mathsf{inv}(B)))))$	zip
	$=_\mathcal{E}$	$\mathsf{take}(d(d(f(a))), \mathsf{zip}(\mathsf{tail}(\mathsf{inv}(B)), \mathsf{tail}(\mathsf{inv}(B))))$	zip
	$=_\mathcal{E}$	$\mathsf{take}(h(d(d(f(a)))), \mathsf{tail}(\mathsf{inv}(B)))$	(2)
	$=_\mathcal{E}$	$\mathsf{take}(d(f(a)), \mathsf{tail}(\mathsf{inv}(B)))$	(1)
	$=_\mathcal{E}$	$\mathsf{take}(d(f(a)), \mathsf{inv}(\mathsf{tail}(B))$	inv
	$=_\mathcal{E}$	$\mathsf{take}(d(f(a)), \mathsf{inv}(\mathsf{zip}(\mathsf{inv}(B), \mathsf{inv}(B))))$	B
	$=_\mathcal{E}$	$\mathsf{take}(d(f(a)), \mathsf{zip}(B, B))$	(3)
	$=_\mathcal{E}$	$\mathsf{take}(h(d(f(a))), B)$	(2)
	$=_\mathcal{E}$	$\mathsf{take}(f(a), B)$	(1)
	$=_\mathcal{E}$	$\mathsf{take}(a, \mathsf{alt})$	*IH:* $\mathsf{take}(f(a), B) = \mathsf{take}(a, \mathsf{alt})$
	$=_\mathcal{E}$	$\mathsf{take}(a, \mathsf{tail}(\mathsf{tail}(\mathsf{alt})))$	alt
	$=_\mathcal{E}$	$\mathsf{take}(s(a), \mathsf{tail}(\mathsf{alt}))$	take
	$=_\mathcal{E}$	$\mathsf{take}(s(s(a)), \mathsf{alt})$	take

Which proves $\mathsf{take}(f(x), B) = \mathsf{take}(x, alt)$.

4.2.3 Strong induction

We can also combine Definition 12 with the strong induction Theorem 8 in the previous chapter to obtain the following method of proving equality of infinite sequences:

Theorem 16 (Strong induction infinite sequences)
Let the set $\mathcal{E}$ of equations over Σ contain the two equations $\mathsf{take}(0, xs) = \mathsf{head}(xs)$ *and* $\mathsf{take}(s(x), xs) = \mathsf{take}(x, \mathsf{tail}(xs))$.

Let $C = \{0, s\} \subseteq \Sigma$ and let $<$ be a well-founded binary relation on $\mathcal{T}(C)$.

Assume that for all $t \in \mathcal{T}(\Sigma)$ of sort natural there exists a term $u \in \mathcal{T}(C)$ such that $u \sim_{\mathcal{E}} t$. Define φ as a function that maps every term in $\mathcal{T}(\Sigma)$ to such a term in $\mathcal{T}(C)$, so $\varphi : \mathcal{T}(\Sigma) \to \mathcal{T}(C)$ such that $\forall t \in \mathcal{T}(\Sigma)$ we have $t \sim_{\mathcal{E}} \varphi(t)$.

Let t,u be terms of sort i satisfying $\mathsf{take}(0, t) =_{\mathcal{E}} \mathsf{take}(0, u)$ *and* $\mathsf{take}(s(a), t) =_{\mathcal{E}'} \mathsf{take}(s(a), u)$ *for $\mathcal{E}' = \mathcal{E} \cup \{\mathsf{take}(v, t) = \mathsf{take}(v, u)\}$, where $v \in \mathcal{T}(\Sigma_{nat}, \{a\})$ such that for all $w \in \mathcal{T}(C)$ $\varphi(v[a := w]) < s(w)$.*

Then $[t] = [u]$.

In some cases this Theorem turns out to be helpful. We will now consider some examples.

Example 11
By adding the previously proved equation $\mathsf{take}(x, \mathsf{zip}(y, y)) = \mathsf{take}(h(x), y)$ *as an equation to $\mathcal{E}$, we can now prove that any element of the sequence* ones *is equal to* 1. *Here we use the strong induction Theorem 16.*

The set of equations $\mathcal{E}$ consists of the following equations:

Number	***Equation***
1	$\mathsf{head}(\mathsf{ones}) = 1$
2	$\mathsf{tail}(\mathsf{ones}) = \mathsf{zip}(\mathsf{ones}, \mathsf{ones})$
3	$\mathsf{take}(x, \mathsf{zip}(y, y)) = \mathsf{take}(h(x), y)$

Goal: $\mathsf{take}(x, \mathsf{ones}) = 1$

$\mathsf{take}(0, \mathsf{ones})$	$=_{\mathcal{E}}$	$\mathsf{head}(\mathsf{ones})$	take
	$=_{\mathcal{E}}$	1	(1)
$\mathsf{take}(s(a), \mathsf{ones})$	$=_{\mathcal{E}}$	$\mathsf{take}(a, \mathsf{tail}(\mathsf{ones}))$	take
	$=_{\mathcal{E}}$	$\mathsf{take}(a, \mathsf{zip}(\mathsf{ones}, \mathsf{ones}))$	(2)
	$=_{\mathcal{E}}$	$\mathsf{take}(h(a), \mathsf{ones})$	(3)
	$=_{\mathcal{E}'}$	1	*IH:* $\mathsf{take}(h(a), \mathsf{ones}) = 1$

Example 12
Now that we have proven $\mathsf{take}(x, ones) = 1$, *we can prove that the infinite sequences* ones *and* $1 : \mathsf{ones}$ *are equal. This is quite interesting, as this originally required the use of Special Contexts. This example also shows that in some cases adding additional equations to $\mathcal{E}$, can enable us to prove goals that use only the original function symbols in $\mathcal{E}$, that we would otherwise not be able to prove.*

Number	***Equation***
1	$\mathsf{head}(\mathsf{ones}) = 1$
2	$\mathsf{tail}(\mathsf{ones}) = \mathsf{zip}(\mathsf{ones}, \mathsf{ones})$
3	$\mathsf{take}(x, \mathsf{zip}(y, y)) = \mathsf{take}(h(x), y)$
4	$\mathsf{take}(x, \mathsf{ones}) = 1$

Goal: $\mathsf{take}(x, \mathsf{ones}) = \mathsf{take}(x, 1 : \mathsf{ones})$

$$
\begin{array}{lll}
\mathsf{take}(0, \mathsf{ones}) & =_{\mathcal{E}} \; 1 & (4) \\
& =_{\mathcal{E}} \; \mathsf{head}(1 : \mathsf{ones}) & : \\
& =_{\mathcal{E}} \; \mathsf{take}(0, 1 : \mathsf{ones}) & \mathsf{take}
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{take}(s(a), \mathsf{ones}) & =_{\mathcal{E}} \; 1 & (4) \\
& =_{\mathcal{E}} \; \mathsf{take}(a, \mathsf{ones}) & (4) \\
& =_{\mathcal{E}} \; \mathsf{take}(a, \mathsf{tail}(1 : \mathsf{ones})) & : \\
& =_{\mathcal{E}} \; \mathsf{take}(s(a), 1 : ones) & \mathsf{take}
\end{array}
$$

Which proves $[\mathsf{ones}] = [1 : \mathsf{ones}]$ *by Theorem 13.*

Sometimes we may be interested in knowing whether equations that describe a certain sequence have a unique solution. In fact, this notion can be used to prove equality in certain cases, as described in [24].

Definition 2 (Uniquely defined)
Let f be a function symbol in Σ of sort i and let $\mathcal{E}$ be a set of equations over Σ. Let $\mathcal{E}' \subseteq \mathcal{E}$ be a set of equations containing f.

Let g be a fresh function symbol of sort i with the same arity and input sorts as f. Let $\mathcal{E}''$ be the set of equations consisting of the equations in $\mathcal{E}'$, with every occurrence of f replaced by g.

Then f is called uniquely defined with respect to the set of equations $\mathcal{E}'$ if $[f(x_1, \ldots, x_n)] =_{\mathcal{E} \cup \mathcal{E}''} [g(x_1, \ldots, x_n)]$.

Example 13
Consider the following definition of A:

$$hd(A) = 0$$
$$\mathsf{take}(s(x), A) = \mathsf{take}(h(x), \mathsf{inv}(A))$$

Now we might be interested in knowing whether these two equations uniquely define A. We can do this by defining B using the same equations, where A is replaced by B.

$$hd(B) = 0$$
$$\mathsf{take}(s(x), B) = \mathsf{take}(h(x), \mathsf{inv}(B))$$

Now, assuming A is uniquely defined by these equations, we can try to prove $[A] = [B]$ using our proving methods. This is indeed the case, as shown by the following proof which uses the strong induction variant as described in Theorem 16. The proof requires the use of an additional lemma. This lemma is part of a class of lemmas that turn out to be helpful in many situations. In Theorem 17 we define this class in more detail and prove that they are valid lemmas.

Number	***Equation***
1	$\mathsf{head}(A) = 0$
2	$\mathsf{take}(s(x), A) = \mathsf{take}(h(x), \mathsf{inv}(A))$
3	$\mathsf{head}(B) = 0$
4	$\mathsf{take}(s(x), B) = \mathsf{take}(h(x), \mathsf{inv}(B))$
5	$\mathsf{take}(x, \mathsf{inv}(y)) = \mathsf{not}(\mathsf{take}(x, y))$

$$
\begin{array}{lll}
\mathsf{take}(0, A) & =_{\mathcal{E}} \; \mathsf{head}(A) & tk \\
& =_{\mathcal{E}} \; 0 & (1) \\
& =_{\mathcal{E}} \; \mathsf{head}(B) & (3) \\
& =_{\mathcal{E}} \; \mathsf{take}(0, B) & \mathsf{take}
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{take}(s(a), A) & =_{\mathcal{E}} \; \mathsf{take}(h(a), \mathsf{inv}(A)) & (2) \\
& =_{\mathcal{E}} \; \mathsf{not}(\mathsf{take}(h(a), A)) & (5) \\
& =_{\mathcal{E}} \; \mathsf{not}(\mathsf{take}(h(a), B)) & \textit{IH: } \mathsf{take}(h(a), A) = \mathsf{take}(h(a), B) \\
& =_{\mathcal{E}} \; \mathsf{take}(h(a), \mathsf{inv}(B)) & (5) \\
& =_{\mathcal{E}} \; \mathsf{take}(s(a), B) & (4)
\end{array}
$$

Which proves $[A] = [B]$ *by Theorem 13.*

In this proof we use the lemma $\mathsf{take}(x, \mathsf{inv}(y)) = \mathsf{not}(\mathsf{take}(x, y))$. It is easy to see that this holds by observing the equations given for inv, since $inv(xs)$ applies the not operator on each element of the sequence xs. The operator take takes the element at a specific position, hence taking an element at position x of the sequence $\mathsf{inv}(xs)$ is equivalent to taking the element at position x of xs, and then applying the not operator. This holds in general for lemmas of a similar shape.

Theorem 17
Let $\mathcal{E}$ be a set of equations over Σ such that $\mathcal{E}$ contains the equations for take*, as defined in Table 4.1. Let $f \in \Sigma$ be a function of sort i which has one or more inputs of sort i, and no inputs of other sorts, so $f : i^k \to i$ for $k \geq 1$. Let f be described by two equations such that:*

- *One of the equations is of the shape $\mathsf{head}(f(x_1, \ldots, x_n)) = g(\mathsf{head}(x_1), \ldots, \mathsf{head}(x_n))$ for variables $x_1, \ldots, x_n$ of sort i and a function $g : d^k \to d \in \Sigma$ where d is the basic data type of the sequence.*
- *The other equation is of the shape $\mathsf{tail}(f(x_1, \ldots, x_n)) = f(\mathsf{tail}(x_1), \ldots, \mathsf{tail}(x_n))$.*

Then $\mathsf{take}(x, f(x_1, \ldots, x_n)) = g(\mathsf{take}(x, x_1), \ldots, \mathsf{take}(x, x_n))$.

Proof. Proof by induction on x. Since x is of type nat, it is either 0 or $s(a)$ for some $a \in \mathcal{T}(\{0, s\})$.

For the base case we have $x = 0$. This yields

$$\begin{array}{lcll} \mathsf{take}(x, f(x_1, \ldots, x_n)) & = & \mathsf{take}(0, f(x_1, \ldots, x_n)) & \\ & =_{\mathcal{E}} & \mathsf{head}(f(x_1, \ldots, x_n)) & \mathsf{take} \\ & =_{\mathcal{E}} & g(\mathsf{head}(x_1), \ldots, \mathsf{head}(x_n)) & f \\ & =_{\mathcal{E}} & g(\mathsf{take}(0, x_1), \ldots, \mathsf{take}(0, x_n)) & \mathsf{take} \\ & = & g(\mathsf{take}(x, x_1), \ldots, \mathsf{take}(x, x_n)) & \end{array}$$

For the other case we have $x = s(a)$ and the induction hypothesis that the Theorem holds for $x = a$. This yields

$$\begin{array}{lcll} \mathsf{take}(x, f(x_1, \ldots, x_n)) & = & \mathsf{take}(s(a), f(x_1, \ldots, x_n)) & \\ & =_{\mathcal{E}} & \mathsf{take}(a, \mathsf{tail}(f(x_1, \ldots, x_n))) & \mathsf{take} \\ & =_{\mathcal{E}} & \mathsf{take}(a, f(\mathsf{tail}(x_1), \ldots, \mathsf{tail}(x_n))) & f \\ & =_{\mathcal{E}} & g(\mathsf{take}(a, \mathsf{tail}(x_1)), \ldots, \mathsf{take}(a, \mathsf{tail}(x_n))) & \mathrm{IH} \\ & =_{\mathcal{E}} & g(\mathsf{take}(s(a), x_1), \ldots, \mathsf{take}(s(a), x_n)) & \mathsf{take} \\ & = & g(\mathsf{take}(x, x_1), \ldots, \mathsf{take}(x, x_n)) & \end{array}$$

Hence in both cases $\mathsf{take}(x, f(x_1, \ldots, x_n)) = g(\mathsf{take}(x, x_1), \ldots, \mathsf{take}(x, x_n))$ holds, which concludes the proof. □

4.3 Special Contexts

The circular coinduction technique can be extended by allowing *special contexts* to be used as additional hypotheses. This is a powerful generalization of Theorem 11: instead of only assuming $\mathsf{fr}(l) = \mathsf{fr}(r)$ we are allowed to assume $\mathsf{fr}(C[l]) = \mathsf{fr}(C[r])$ for special contexts C.

Informally, a context is called special if the n-th element of the infinite sequence represented by $C[s]$ only depends on the first n elements of s. This notion originates from [17]. Using our notation, we can define special contexts as follows based on the definition given in [24]:

Definition 3
Context C of sort i with the hole of sort i is called special with respect to a set of equations $\mathcal{E}$ over Σ if and only if for every $n \geq 0$ the following holds: for any terms $t, t' \in \mathcal{T}(\Sigma, \mathcal{X})_i$, $\mathsf{take}(n, C[t]) =_{\mathcal{E}} \mathsf{take}(n, C[t'])$ if $\mathsf{take}(x, t) = \mathsf{take}(x, t')$ for all $x \leq n$.

This definition is obtained by replacing every occurrence of $[t](k)$ in the original definition by $\mathsf{take}(k, t)$.

According to this definition, for example the empty context $\Box$ is considered a special context: the empty context applied to any infinite sequence yields the sequence itself, therefore clearly the condition for $\Box$ to be a special context holds. Some other examples of special contexts are $\mathsf{zip}(xs, *)$, $\mathsf{zip}(*, xs)$, and $\mathsf{inv}(*)$.

An example of a context that is not special is $\mathsf{tail}(*)$. This can easily be observed since the n-th element of $\mathsf{tail}(xs)$ is the $n+1$-th element of xs.

Using these special contexts to create additional hypotheses does not affect the soundness of the proof system. The following Theorem describes circular coinduction extended with special contexts.

Theorem 18 (Circular Coinduction with Special Contexts [24])
Let $\mathcal{E}$ be a set of equations over Σ not containing the symbol fr and the sort Frozen. For each sort s that occurs in Σ, add the symbol $\mathsf{fr} : s \to Frozen$. Let Ω be a set of special contexts.
Assume that

$$\mathsf{head}(t) =_{\mathcal{E}} \mathsf{head}(u) \text{ and } \mathsf{fr}(\mathsf{tail}(t)) =_{\mathcal{E}'} \mathsf{fr}(\mathsf{tail}(u))$$

for $\mathcal{E}' = \mathcal{E} \cup \bigcup_{C \in \Omega} \{\mathsf{fr}(C[t]) = \mathsf{fr}(C[u])\}$.
Then $[t] = [u]$.

We have shown before that circular coinduction is equivalent to the induction approach with the take operator as described in Theorem 13. Hence we can create a modification of Theorem 13 that uses special contexts.

Theorem 19 (Basic induction (see Theorem 13) with special contexts)
Let the set $\mathcal{E}$ of equations contain the two equations $\mathsf{take}(0, xs) = \mathsf{head}(xs)$ and $\mathsf{take}(s(x), xs) = \mathsf{take}(x, \mathsf{tail}(xs))$. Let t,u be terms of sort i satisfying $\mathsf{take}(0, t) =_{\mathcal{E}} \mathsf{take}(0, u)$ and $\mathsf{take}(s(a), t) =_{\mathcal{E}'} \mathsf{take}(s(a), u)$ for $\mathcal{E}' = \mathcal{E} \cup \bigcup_{C \in \Omega} \{\mathsf{take}(a, C(t)) = \mathsf{take}(a, C(u))\}$, where Ω is a set of special contexts.
Then $[t] = [u]$.

Since the empty context is a special context by definition, this extension with special contexts can be seen as a generalization of Theorem 13. Since the empty context is a special context by definition, Theorem 13 is an instance of Theorem 19, obtained by. This makes the extension with special contexts a more powerful proof system, as it can prove any example the original method can prove and we will show there are examples which can only be proved using the extended version with special contexts.

For example, it is not possible to prove $\mathsf{ones} = 1 : \mathsf{ones}$ where ones is defined by $\mathsf{ones} = 1 : \mathsf{zip}(\mathsf{ones}, \mathsf{ones})$ and $\mathsf{zip}(x : xs, ys) = x : \mathsf{zip}(ys, xs)$ using the standard circular coinduction approach.

Proving the first requirement is trivial, however proving the second requirement results in a failure using the standard circular coinduction approach. Since $\mathsf{zip}(*, x)$ is a special context we may use the hypothesis $\mathsf{fr}(\mathsf{zip}(\mathsf{ones}, x)) = \mathsf{fr}(\mathsf{zip}(1 : \mathsf{ones}, x))$ which allows us to prove the equation

as follows:

$$\begin{array}{lll}
\mathsf{take}(s(a), \mathsf{ones}) & =_{\mathcal{E}} \; \mathsf{take}(a, \mathsf{tail}(\mathsf{ones})) & \mathsf{take} \\
& =_{\mathcal{E}} \; \mathsf{take}(a, \mathsf{tail}(1 : \mathsf{zip}(\mathsf{ones}, \mathsf{ones}))) & \mathsf{ones} \\
& =_{\mathcal{E}} \; \mathsf{take}(a, \mathsf{zip}(\mathsf{ones}, \mathsf{ones})) & : \\
& =_{\mathcal{E}'} \; \mathsf{take}(a, \mathsf{zip}(1 : \mathsf{ones}, \mathsf{ones})) & \text{IH: } \mathsf{take}(a, C[\mathsf{ones}]) = \mathsf{take}(a, C[1 : \mathsf{ones}]) \\
& =_{\mathcal{E}} \; \mathsf{take}(a, 1 : \mathsf{zip}(\mathsf{ones}, \mathsf{ones})) & \mathsf{zip} \\
& =_{\mathcal{E}} \; \mathsf{take}(a, \mathsf{ones}) & \mathsf{ones} \\
& =_{\mathcal{E}} \; \mathsf{take}(a, \mathsf{tail}(1 : \mathsf{ones})) & : \\
& =_{\mathcal{E}} \; \mathsf{take}(s(a), 1 : \mathsf{ones}) & \mathsf{take}
\end{array}$$

4.4 Morphic sequences

An interesting class of non-periodic infinite sequences are *morphic* sequences [1, 2, 21]. These are sequences that arise from the (infinite) iteration of a morphism. They appear in many different places in mathematics and computer science.

A morphic sequence is a homomorphic image of a *pure morphic* sequence, which can be described as follows. Let Σ^* denote the set of all finite words over Σ. Let $h : \Sigma^* \to \Sigma^*$ be a morphism, that is, a map which satisfies $h(xy) = h(x)h(y)$ for all $x, y \in \Sigma^*$. Then it suffices to define h on each element of Σ. If there exists $a \in \Sigma$ and $t \in \Sigma^*$ such that

1. $h(a) = at$; and
2. $h^i(t) \neq \epsilon$ for all $i \geq 0$,

then h is called *prolongable* on a. Now, iterating h on a produces a sequence of words of increasing length

$$a, h(a), h^2(a), \ldots$$

where each word is a prefix of the following word. If we iterate infinitely often, this produces the infinite sequence

$$h^\omega(a) = ath(t)h^2(t)h^3(t)\ldots$$

which we call a pure morphic sequence.

4.4.1 Thue-Morse sequence

An example of a morphic sequence is the *Thue-Morse* sequence [2]. This sequence is defined by $f^\omega(0)$ where $f(0) = 01$ and $f(1) = 10$. Iterating f on 0 produces:

$$0, f(0) = 01, f^2(0) = 0110, f^3(0) = 01101001, \ldots$$

Which yields the following infinite sequence:

$$\mathsf{morse} = 0 : 1 : 1 : 0 : 1 : 0 : 0 : 1 : 0 : 0 : 1 : 0 : 1 : 1 : \ldots$$

This well-known sequence has several alternative definitions and characterizations. We focus on the following definition of this sequence:

$$\mathsf{head}(\mathsf{morse}) = 0,\ \mathsf{tail}(\mathsf{morse}) = \mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse}))$$

where $\mathsf{head}, \mathsf{tail}, \mathsf{zip}$ and inv are defined in the same way as before. We can specify f as follows:

$$\mathsf{head}(f(x)) = \mathsf{head}(x),\ \mathsf{head}(\mathsf{tail}(f(x))) = \mathsf{not}(\mathsf{head}(x)),\ \mathsf{tail}(\mathsf{tail}(f(x))) = f(\mathsf{tail}(x))$$

Now we can prove that $[f(\mathsf{morse})] = [\mathsf{morse}]$, which means that morse is a fixed point of f. As f only has one fixed point starting in 0, this proves that this alternative definition defines the same sequence as $f^\omega(0)$.

Example 14 (Morse fixpoint)
Our tool first proves $\mathsf{take}(x, \mathsf{zip}(y, \mathsf{inv}(y))) = \mathsf{take}(x, f(y))$ *(for the proof see Appendix B) and then uses this equation as a lemma to prove (1)* $\mathsf{take}(x, \mathsf{morse}) = \mathsf{take}(x, f(\mathsf{morse}))$.

$$
\begin{array}{lll}
\mathsf{take}(0, f(\mathsf{morse})) & =_{\mathcal{E}} \mathsf{head}(f(\mathsf{morse})) & \mathsf{take} \\
& =_{\mathcal{E}} \mathsf{head}(\mathsf{morse}) & f \\
& =_{\mathcal{E}} \mathsf{take}(0, \mathsf{morse}) & \mathsf{take} \\
\\
\mathsf{take}(s(a), f(\mathsf{morse})) & =_{\mathcal{E}} \mathsf{take}(s(a), \mathsf{zip}(\mathsf{morse}, \mathsf{inv}(\mathsf{morse}))) & (1) \\
& =_{\mathcal{E}} \mathsf{take}(a, \mathsf{tail}(\mathsf{zip}(\mathsf{morse}, \mathsf{inv}(\mathsf{morse})))) & \mathsf{take} \\
& =_{\mathcal{E}} \mathsf{take}(a, \mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse}))) & \mathsf{zip} \\
& =_{\mathcal{E}} \mathsf{take}(a, \mathsf{tail}(\mathsf{morse})) & \mathsf{morse} \\
& =_{\mathcal{E}} \mathsf{take}(s(a), \mathsf{morse}) & \mathsf{take}
\end{array}
$$

Which proves $[\mathsf{morse})] = [f(\mathsf{morse}))]$ *by Theorem 13.*

The Thue-Morse sequence has some interesting properties. For example, it is non-periodic. A sequence is called periodic if it repeats itself after some number of elements. More specifically a sequence $a_0, a_1, a_2, \ldots$ is called periodic if $a_n = a_{n+p}$ for all n and some p. It is also self-similar, which roughly speaking means that a part of the sequence contains a copy of the sequence itself. In this case, taking the even elements of morse $(0, 2, 4, \ldots)$ yields the sequence morse itself.

Example 15 (Even morse)
We can also prove this. We first prove automatically that (1) $\mathsf{take}(x, \mathsf{even}(\mathsf{zip}(xs, ys))) = \mathsf{take}(x, xs)$ *(proof in Appendix B) and using this lemma we can prove* $\mathsf{take}(x, \mathsf{even}(\mathsf{morse})) = \mathsf{take}(x, \mathsf{morse})$.

$$
\begin{array}{lll}
\mathsf{take}(0, \mathsf{even}(\mathsf{morse})) & =_{\mathcal{E}} \mathsf{head}(\mathsf{even}(\mathsf{morse})) & \mathsf{take} \\
& =_{\mathcal{E}} \mathsf{head}(\mathsf{morse}) & \mathsf{even} \\
& =_{\mathcal{E}} \mathsf{take}(0, \mathsf{morse}) & \mathsf{take} \\
\\
\mathsf{take}(s(a), \mathsf{even}(\mathsf{morse})) & =_{\mathcal{E}} \mathsf{take}(a, \mathsf{tail}(\mathsf{even}(\mathsf{morse}))) & \mathsf{take} \\
& =_{\mathcal{E}} \mathsf{take}(a, \mathsf{even}(\mathsf{tail}(\mathsf{tail}(\mathsf{morse})))) & \mathsf{even} \\
& =_{\mathcal{E}} \mathsf{take}(a, \mathsf{even}(\mathsf{tail}(\mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse})))))) & \mathsf{morse} \\
& =_{\mathcal{E}} \mathsf{take}(a, \mathsf{even}(\mathsf{zip}(\mathsf{tail}(\mathsf{morse}), \mathsf{tail}(\mathsf{inv}(\mathsf{morse})))))) & \mathsf{zip} \\
& =_{\mathcal{E}} \mathsf{take}(a, \mathsf{tail}(\mathsf{morse})) & (1) \\
& =_{\mathcal{E}} \mathsf{take}(s(a), \mathsf{morse}) & \mathsf{take}
\end{array}
$$

4.4.2 Period doubling sequence

A second example of a morphic sequence is the period doubling sequence (pd) [2]. This can be defined as $f^{\omega}(0)$ where $f(0) = 01$ and $f(1) = 00$. This describes the following sequence:

$$\mathsf{pd} = 0 : 1 : 0 : 0 : 0 : 1 : 0 : 1 : 0 : 1 : 0 : 0 : 0 : 1 : \ldots$$

An alternative characterization can be obtained from morse in the following way: compare the first and second element of morse, if they are equal then output 1, otherwise 0. Continue with the second and third elements of morse, and so on. In order to accomplish this we define the operator iff on booleans as follows:

$$
\begin{aligned}
\mathsf{iff}(0, 0) &= 1 \\
\mathsf{iff}(0, 1) &= 0 \\
\mathsf{iff}(1, 0) &= 0 \\
\mathsf{iff}(1, 1) &= 1
\end{aligned}
$$

Now if we apply this as described on morse, we obtain:

$$\begin{aligned}
\mathsf{iff}(\mathsf{morse}_0, \mathsf{morse}_1) &= 0 \\
\mathsf{iff}(\mathsf{morse}_1, \mathsf{morse}_2) &= 1 \\
\mathsf{iff}(\mathsf{morse}_2, \mathsf{morse}_3) &= 0 \\
\mathsf{iff}(\mathsf{morse}_3, \mathsf{morse}_4) &= 0 \\
\mathsf{iff}(\mathsf{morse}_4, \mathsf{morse}_5) &= 0 \\
\mathsf{iff}(\mathsf{morse}_5, \mathsf{morse}_6) &= 1 \\
\mathsf{iff}(\mathsf{morse}_6, \mathsf{morse}_7) &= 0 \\
\mathsf{iff}(\mathsf{morse}_7, \mathsf{morse}_8) &= 1 \\
&\dots
\end{aligned}$$

which seems to match the sequence pd as expected. To apply this on an infinite sequence we define the operator $\mathsf{diff} : i \to i$ as follows:

$$\mathsf{head}(\mathsf{diff}(x)) = \mathsf{iff}(\mathsf{head}(x),\ \mathsf{head}(\mathsf{tail}(x))),\ \mathsf{tail}(\mathsf{diff}(x)) = \mathsf{diff}(\mathsf{tail}(x))$$

Therefore, an alternative definition of pd is the following:

$$\mathsf{pd} = \mathsf{diff}(\mathsf{morse})$$

We can define f as follows:

$$\mathsf{head}(f(x)) = 0,\ \mathsf{head}(\mathsf{tail}(f(x))) = \mathsf{not}(\mathsf{head}(x)),\ \mathsf{tail}(\mathsf{tail}(f(x))) = f(\mathsf{tail}(x))$$

Example 16 (Period doubling fixpoint)
Similar to what we did before with morse*, we can now prove* $[f(\mathsf{pd})] = [\mathsf{pd}]$*, proving that* pd *is a fixed point of* f*, and as* f *only has one fixed point* $f^\omega(0)$ *starting in 0, this proves that these two characterizations define the same sequence.*

In order to prove $[f(\mathsf{pd})] = [\mathsf{pd}]$ *the following lemmas were required, for which the proofs can be found in Appendix B:*

Number	***Equation***
1	$\mathsf{diff}(\mathsf{zip}(x, \mathsf{inv}(x))) = \mathsf{zip}(\mathsf{zeros}, \mathsf{inv}(\mathsf{diff}(x)))$
2	$f(x) = \mathsf{zip}(\mathsf{zeros}, \mathsf{inv}(x))$

These lemmas were proven automatically by our tool. After proving these lemmas the tool was able automatically prove $[f(\mathsf{pd})] = [\mathsf{pd}]$ *and gave the following conversions as output:*

$\mathsf{take}(0, \mathsf{pd})$	$=_\mathcal{E}$	$\mathsf{take}(0, \mathsf{diff}(\mathsf{morse}))$	pd
	$=_\mathcal{E}$	$\mathsf{head}(\mathsf{diff}(\mathsf{morse}))$	take
	$=_\mathcal{E}$	$\mathsf{iff}(\mathsf{head}(\mathsf{morse}), \mathsf{head}(\mathsf{tail}(\mathsf{morse})))$	diff
	$=_\mathcal{E}$	$\mathsf{iff}(\mathsf{head}(\mathsf{morse}), \mathsf{head}(\mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse}))))$	morse
	$=_\mathcal{E}$	$\mathsf{iff}(\mathsf{head}(\mathsf{morse}), \mathsf{head}(\mathsf{inv}(\mathsf{morse})))$	zip
	$=_\mathcal{E}$	$\mathsf{iff}(\mathsf{head}(\mathsf{morse}), \mathsf{not}(\mathsf{head}(\mathsf{morse})))$	inv
	$=_\mathcal{E}$	$\mathsf{iff}(0, \mathsf{not}(0))$	morse
	$=_\mathcal{E}$	$\mathsf{iff}(0, 1)$	not
	$=_\mathcal{E}$	0	iff
	$=_\mathcal{E}$	$\mathsf{head}(f(\mathsf{pd}))$	f
	$=_\mathcal{E}$	$\mathsf{take}(0, f(\mathsf{pd}))$	take

$$
\begin{array}{lll}
\mathsf{take}(s(0), \mathsf{pd}) & =_{\mathcal{E}} \mathsf{take}(s(0), \mathsf{diff}(\mathsf{morse})) & \mathsf{pd} \\
& =_{\mathcal{E}} \mathsf{take}(0, \mathsf{tail}(\mathsf{diff}(\mathsf{morse}))) & \mathsf{take} \\
& =_{\mathcal{E}} \mathsf{take}(0, \mathsf{diff}(\mathsf{tail}(\mathsf{morse}))) & \mathsf{diff} \\
& =_{\mathcal{E}} \mathsf{head}(\mathsf{diff}(\mathsf{tail}(\mathsf{morse}))) & \mathsf{take} \\
& =_{\mathcal{E}} \mathsf{head}(\mathsf{diff}(\mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse})))) & \mathsf{morse} \\
& =_{\mathcal{E}} \mathsf{iff}(\mathsf{head}(\mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse}))), \mathsf{head}(\mathsf{tail}(\mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse}))))) & \mathsf{diff} \\
& =_{\mathcal{E}} \mathsf{iff}(\mathsf{head}(\mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse}))), \mathsf{head}(\mathsf{zip}(\mathsf{tail}(\mathsf{morse}), \mathsf{tail}(\mathsf{inv}(\mathsf{morse}))))) & \mathsf{zip} \\
& =_{\mathcal{E}} \mathsf{iff}(\mathsf{head}(\mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse})))), & \\
& \quad \mathsf{head}(\mathsf{zip}(\mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse})), \mathsf{tail}(\mathsf{inv}(\mathsf{morse}))))) & \mathsf{morse} \\
& =_{\mathcal{E}} \mathsf{iff}(\mathsf{head}(\mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse})))), & \\
& \quad \mathsf{head}(\mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse})))) & \mathsf{zip} \\
& =_{\mathcal{E}} \mathsf{iff}(\mathsf{head}(\mathsf{inv}(\mathsf{morse})), \mathsf{head}(\mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse})))) & \mathsf{zip} \\
& =_{\mathcal{E}} \mathsf{iff}(\mathsf{head}(\mathsf{inv}(\mathsf{morse})), \mathsf{head}(\mathsf{inv}(\mathsf{morse}))) & \mathsf{zip} \\
& =_{\mathcal{E}} \mathsf{iff}(\mathsf{not}(\mathsf{head}(\mathsf{morse})), \mathsf{not}(\mathsf{head}(\mathsf{morse}))) & \mathsf{inv} \\
& =_{\mathcal{E}} \mathsf{iff}(\mathsf{not}(0), \mathsf{not}(0)) & \mathsf{morse} \\
& =_{\mathcal{E}} \mathsf{iff}(1, 1) & \mathsf{not} \\
& =_{\mathcal{E}} 1 & \mathsf{iff} \\
& =_{\mathcal{E}} \mathsf{not}(0) & \mathsf{not} \\
& =_{\mathcal{E}} \mathsf{not}(\mathsf{iff}(0, 1)) & \mathsf{iff} \\
& =_{\mathcal{E}} \mathsf{not}(\mathsf{iff}(0, \mathsf{not}(0))) & \mathsf{not} \\
& =_{\mathcal{E}} \mathsf{not}(\mathsf{iff}(\mathsf{head}(\mathsf{morse}), \mathsf{not}(\mathsf{head}(\mathsf{morse})))) & \mathsf{morse} \\
& =_{\mathcal{E}} \mathsf{not}(\mathsf{iff}(\mathsf{head}(\mathsf{morse}), \mathsf{head}(\mathsf{inv}(\mathsf{morse})))) & \mathsf{inv} \\
& =_{\mathcal{E}} \mathsf{not}(\mathsf{iff}(\mathsf{head}(\mathsf{morse}), \mathsf{head}(\mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse})))))) & \mathsf{zip} \\
& =_{\mathcal{E}} \mathsf{not}(\mathsf{iff}(\mathsf{head}(\mathsf{morse}), \mathsf{head}(\mathsf{tail}(\mathsf{morse})))) & \mathsf{morse} \\
& =_{\mathcal{E}} \mathsf{not}(\mathsf{head}(\mathsf{diff}(\mathsf{morse}))) & \mathsf{diff} \\
& =_{\mathcal{E}} \mathsf{not}(\mathsf{head}(\mathsf{pd})) & \mathsf{pd} \\
& =_{\mathcal{E}} \mathsf{head}(\mathsf{tail}(f(\mathsf{pd}))) & f \\
& =_{\mathcal{E}} \mathsf{take}(0, \mathsf{tail}(f(\mathsf{pd}))) & \mathsf{take} \\
& =_{\mathcal{E}} \mathsf{take}(s(0), f(\mathsf{pd})) & \mathsf{take}
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{take}(s(s(a)), \mathsf{pd}) & =_{\mathcal{E}} \mathsf{take}(s(s(a)), \mathsf{diff}(\mathsf{morse})) & \mathsf{pd} \\
& =_{\mathcal{E}} \mathsf{take}(s(a), \mathsf{tail}(\mathsf{diff}(\mathsf{morse}))) & \mathsf{take} \\
& =_{\mathcal{E}} \mathsf{take}(s(a), \mathsf{diff}(\mathsf{tail}(\mathsf{morse}))) & \mathsf{diff} \\
& =_{\mathcal{E}} \mathsf{take}(s(a), \mathsf{diff}(\mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse})))) & \mathsf{morse} \\
& =_{\mathcal{E}} \mathsf{take}(a, \mathsf{tail}(\mathsf{diff}(\mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse}))))) & \mathsf{take} \\
& =_{\mathcal{E}} \mathsf{take}(a, \mathsf{diff}(\mathsf{tail}(\mathsf{zip}(\mathsf{inv}(\mathsf{morse}), \mathsf{tail}(\mathsf{morse}))))) & \mathsf{diff} \\
& =_{\mathcal{E}} \mathsf{take}(a, \mathsf{diff}(\mathsf{zip}(\mathsf{tail}(\mathsf{morse}), \mathsf{tail}(\mathsf{inv}(\mathsf{morse}))))) & \mathsf{zip} \\
& =_{\mathcal{E}} \mathsf{take}(a, \mathsf{diff}(\mathsf{zip}(\mathsf{tail}(\mathsf{morse}), \mathsf{inv}(\mathsf{tail}(\mathsf{morse})))))) & \mathsf{inv} \\
& =_{\mathcal{E}} \mathsf{take}(a, \mathsf{zip}(\mathsf{zeros}, \mathsf{inv}(\mathsf{diff}(\mathsf{tail}(\mathsf{morse})))))) & (1) \\
& =_{\mathcal{E}} \mathsf{take}(a, \mathsf{zip}(\mathsf{zeros}, \mathsf{inv}(\mathsf{tail}(\mathsf{diff}(\mathsf{morse})))))) & \mathsf{diff} \\
& =_{\mathcal{E}} \mathsf{take}(a, f(\mathsf{tail}(\mathsf{diff}(\mathsf{morse})))) & (2) \\
& =_{\mathcal{E}} \mathsf{take}(a, f(\mathsf{tail}(\mathsf{pd}))) & \mathsf{pd} \\
& =_{\mathcal{E}} \mathsf{take}(a, \mathsf{tail}(\mathsf{tail}(f(\mathsf{pd})))) & f \\
& =_{\mathcal{E}} \mathsf{take}(s(a), \mathsf{tail}(f(\mathsf{pd}))) & \mathsf{take} \\
& =_{\mathcal{E}} \mathsf{take}(s(s(a)), f(\mathsf{pd})) & \mathsf{take}
\end{array}
$$

4.5 Infinite sequences of natural numbers

So far all of the examples we have considered were infinite sequences of type Boolean. However, our techniques can be applied to infinite sequence of any data type. An example of a different type of infinite sequences are infinite sequences of natural numbers. In this section we will apply our techniques to prove equalities on such sequences.

Example 17
Consider the function $N : nat \to i$ which describes the infinite sequence $N(x) = x, x+1, x+2, \ldots$. We can describe this function with the following equations:

$$\mathsf{head}(N(x)) = x,\ \mathsf{tail}(N(x)) = N(s(x))$$

Secondly, we define the function $S : i \to i$ which applies the operator s on all elements of a sequence. We can describe this function as follows:

$$\mathsf{head}(S(xs)) = s(\mathsf{head}(xs)),\ \mathsf{tail}(S(xs)) = S(\mathsf{tail}(xs))$$

Now the tail *of $N(x)$ should be equal to $S(N(x))$. We can prove this as follows:*

$\mathsf{take}(0, \mathsf{tail}(N(y)))$	$=_{\mathcal{E}}$	$\mathsf{take}(0, \mathsf{tail}(N(\mathsf{head}(N(y)))))$	N
	$=_{\mathcal{E}}$	$\mathsf{take}(0, N(s(\mathsf{head}(N(y)))))$	N
	$=_{\mathcal{E}}$	$\mathsf{take}(0, N(\mathsf{head}(S(N(y)))))$	N
	$=_{\mathcal{E}}$	$\mathsf{head}(N(\mathsf{head}(S(N(y)))))$	take
	$=_{\mathcal{E}}$	$\mathsf{head}(N(\mathsf{take}(0, S(N(y)))))$	take
	$=_{\mathcal{E}}$	$\mathsf{take}(0, S(N(y)))$	take
$\mathsf{take}(s(a), \mathsf{tail}(N(y)))$	$=_{\mathcal{E}}$	$\mathsf{take}(s(a), N(s(y)))$	N
	$=_{\mathcal{E}}$	$\mathsf{take}(a, \mathsf{tail}(N(s(y))))$	take
	$=_{\mathcal{E}}$	$\mathsf{take}(a, S(N(s(y))))$	*IH:* $\mathsf{take}(a, \mathsf{tail}(N(y))) = \mathsf{take}(a, S(N(y)))$
	$=_{\mathcal{E}}$	$\mathsf{take}(a, S(\mathsf{tail}(N(y)))))$	N
	$=_{\mathcal{E}}$	$\mathsf{take}(a, \mathsf{tail}(S(N(y)))))$	S
	$=_{\mathcal{E}}$	$\mathsf{take}(s(a), S(N(y)))$	take

Which proves $[\mathsf{tail}(N(x))] = [S(N(x))]$ by Theorem 13

Example 18
Similarly to how we defined the S operator on infinite sequences, we can also define the operator plus*, which applies $+$ pairwise on the elements of two infinite sequences. We can describe* plus *as follows:*

$$\mathsf{head}(\mathsf{plus}(xs, ys)) = +(\mathsf{head}(xs), \mathsf{head}(ys)),\ \mathsf{tail}(\mathsf{plus}(xs, ys)) = \mathsf{plus}(\mathsf{tail}(xs), \mathsf{tail}(ys))$$

Here we use associativity of $+$ as a lemma: $+(x, +(y, z)) = +(+(x, y), z)$ (1). This allows us to automatically prove $[\mathsf{plus}(x, \mathsf{plus}(y, z))] = [\mathsf{plus}(\mathsf{plus}(x, y), z)]$ using our tool. The proof of this lemma can be found in Appendix B

$$
\begin{array}{lll}
\mathsf{take}(0, \mathsf{plus}(x, \mathsf{plus}(y,z))) & =_{\mathcal{E}} \; \mathsf{head}(\mathsf{plus}(x, \mathsf{plus}(y,z))) & \mathsf{take} \\
& =_{\mathcal{E}} \; +(\mathsf{head}(x), \mathsf{head}(\mathsf{plus}(y,z))) & \mathsf{plus} \\
& =_{\mathcal{E}} \; +(\mathsf{head}(x), +(\mathsf{head}(y), \mathsf{head}(z))) & \mathsf{plus} \\
& =_{\mathcal{E}} \; +(+(\mathsf{head}(x), \mathsf{head}(y)), \mathsf{head}(z)) & (1) \\
& =_{\mathcal{E}} \; +(\mathsf{head}(\mathsf{plus}(x,y)), \mathsf{head}(z)) & \mathsf{plus} \\
& =_{\mathcal{E}} \; \mathsf{head}(\mathsf{plus}(\mathsf{plus}(x,y), z)) & \mathsf{plus} \\
& =_{\mathcal{E}} \; \mathsf{take}(0, \mathsf{plus}(\mathsf{plus}(x,y), z)) & \mathsf{take} \\
\\
\mathsf{take}(s(a), \mathsf{plus}(x, \mathsf{plus}(y,z))) & =_{\mathcal{E}} \; \mathsf{take}(a, \mathsf{tail}(\mathsf{plus}(x, \mathsf{plus}(y,z)))) & \mathsf{take} \\
& =_{\mathcal{E}} \; \mathsf{take}(a, \mathsf{plus}(\mathsf{tail}(x), \mathsf{tail}(\mathsf{plus}(y,z)))) & \mathsf{plus} \\
& =_{\mathcal{E}} \; \mathsf{take}(a, \mathsf{plus}(\mathsf{tail}(x), \mathsf{plus}(\mathsf{tail}(y), \mathsf{tail}(z)))) & \mathsf{plus} \\
& =_{\mathcal{E}} \; \mathsf{take}(a, \mathsf{plus}(\mathsf{plus}(\mathsf{tail}(x), \mathsf{tail}(y)), \mathsf{tail}(z))) & \mathit{IH} \\
& =_{\mathcal{E}} \; \mathsf{take}(a, \mathsf{plus}(\mathsf{tail}(\mathsf{plus}(x,y)), \mathsf{tail}(z))) & \mathsf{plus} \\
& =_{\mathcal{E}} \; \mathsf{take}(a, \mathsf{tail}(\mathsf{plus}(\mathsf{plus}(x,y), z))) & \mathsf{plus} \\
& =_{\mathcal{E}} \; \mathsf{take}(s(a), \mathsf{plus}(\mathsf{plus}(x,y), z)) & \mathsf{take}
\end{array}
$$

Which proves the equation by Theorem 13.

Example 19
We can now also prove commutativity of plus*. This requires the lemma that $+$ is commutative: $+(x,y) = +(y,x)$. We can prove this using our tool, see Appendix B. The same definition of* plus *is used as before.*

$$
\begin{array}{lll}
\mathsf{take}(0, \mathsf{plus}(y,z)) & =_{\mathcal{E}} \; \mathsf{head}(\mathsf{plus}(y,z)) & \mathsf{take} \\
& =_{\mathcal{E}} \; +(\mathsf{head}(y), \mathsf{head}(z)) & \mathsf{plus} \\
& =_{\mathcal{E}} \; +(\mathsf{head}(z), \mathsf{head}(y)) & (1) \\
& =_{\mathcal{E}} \; \mathsf{head}(\mathsf{plus}(z,y)) & \mathsf{plus} \\
& =_{\mathcal{E}} \; \mathsf{take}(0, \mathsf{plus}(z,y)) & \mathsf{take} \\
\\
\mathsf{take}(s(a), \mathsf{plus}(y,z)) & =_{\mathcal{E}} \; \mathsf{take}(a, \mathsf{tail}(\mathsf{plus}(y,z))) & \mathsf{take} \\
& =_{\mathcal{E}} \; \mathsf{take}(a, \mathsf{plus}(\mathsf{tail}(y), \mathsf{tail}(z))) & \mathsf{plus} \\
& =_{\mathcal{E}} \; \mathsf{take}(a, \mathsf{plus}(\mathsf{tail}(z), \mathsf{tail}(y))) & \mathit{IH} \\
& =_{\mathcal{E}} \; \mathsf{take}(a, \mathsf{tail}(\mathsf{plus}(z,y))) & \mathsf{plus} \\
& =_{\mathcal{E}} \; \mathsf{take}(s(a), \mathsf{plus}(z,y)) & \mathsf{take}
\end{array}
$$

Which proves $[\mathsf{plus}(x,y)] = [\mathsf{plus}(y,x)]$ by Theorem 13.

In a similar way we can also prove properties like commutativity and associativity of other common functions on natural numbers, such as multiplication or subtraction.

We can also prove properties of well known sequences of natural numbers. For example, the Fibonacci sequence formed by $F_0 = 0,\ F_1 = 1,\ F_n = F_{n-1} = F_{n-2}$ for $n > 1$:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, \ldots$$

Example 20
Let F_n denote the n-th Fibonacci number. Then the sum of all Fibonacci numbers $F_0, F_1, \ldots, F_n$ is equal to $F_{n+2} - 1$. More precisely:

$$\sum_{i=0}^{n} F_i = F_{n+2} - 1$$

or equivalently, after adding one to both sides

$$1 + \sum_{i=0}^{n} F_i = F_{n+2}$$

We can prove that this holds for the whole Fibonacci sequence. That is, if we take the Fibonacci sequence starting from the first element and add one to each element, this should be equal to the Fibonacci sequence starting from the second element.

First we define the Fibonacci sequence as follows:

$$\begin{aligned}\mathsf{head}(fibseq) &= 0\\ \mathsf{head}(\mathsf{tail}(fibseq)) &= s(0)\\ \mathsf{tail}(\mathsf{tail}(fibseq)) &= \mathsf{plus}(fibseq, \mathsf{tail}(fibseq)\end{aligned}$$

Next we define the operator $\mathsf{acc} : i \to i$*. This operator has as input an infinite sequence of natural numbers and creates a new infinite sequence such that the* n*-th element is the sum of the elements at positions* $0, 1, \ldots, n$ *of the original sequence. Hence,* $\mathsf{acc}(xs)(n) = \sum_{i=0}^{n} xs(i)$*. This operator can be defined by the following equations:*

$$\begin{aligned}\mathsf{head}(\mathsf{acc}(x)) &= \mathsf{head}(x)\\ \mathsf{take}(\mathsf{acc}(x)) &= \mathsf{plus}(\mathsf{acc}(x), \mathsf{take}(x))\end{aligned}$$

To add 1 to each element of $fibseq$*, we use the* S *operator we defined earlier, which applies the* s *function on each element of an infinite sequence.*

Now we can prove that $[S(\mathsf{acc}(fibseq))] = [\mathsf{tail}(\mathsf{tail}(fibseq))]$ *using some lemmas. Lemma* (2) *was proven above, and Lemma* (1) *and* (3) *follow from Theorem 17.*

Number	***Equation***
1	$\mathsf{take}(x, S(xs)) = s(\mathsf{take}(x, xs))$
2	$\mathsf{plus}(x, y) = \mathsf{plus}(y, x)$
3	$+(\mathsf{take}(x, xs), \mathsf{take}(x, ys)) = \mathsf{take}(x, \mathsf{plus}(xs, ys))$

$$\begin{array}{lcll}
\mathsf{take}(0, S(\mathsf{acc}(fibseq))) & =_{\mathcal{E}} & s(\mathsf{take}(0, \mathsf{acc}(fibseq))) & (1)\\
& =_{\mathcal{E}} & s(+(0, \mathsf{take}(0, \mathsf{acc}(fibseq)))) & +\\
& =_{\mathcal{E}} & s(+(0, \mathsf{head}(\mathsf{acc}(fibseq)))) & \mathsf{take}\\
& =_{\mathcal{E}} & s(+(0, \mathsf{head}(fibseq))) & \mathsf{acc}\\
& =_{\mathcal{E}} & +(s(0), \mathsf{head}(fibseq)) & +\\
& =_{\mathcal{E}} & +(\mathsf{head}(\mathsf{tail}(fibseq)), \mathsf{head}(fibseq)) & fibseq\\
& =_{\mathcal{E}} & \mathsf{head}(\mathsf{plus}(\mathsf{tail}(fibseq), fibseq)) & \mathsf{plus}\\
& =_{\mathcal{E}} & \mathsf{take}(0, \mathsf{plus}(\mathsf{tail}(fibseq), fibseq)) & \mathsf{take}\\
& =_{\mathcal{E}} & \mathsf{take}(0, \mathsf{plus}(fibseq, \mathsf{tail}(fibseq))) & (2)\\
& =_{\mathcal{E}} & \mathsf{take}(0, \mathsf{tail}(\mathsf{tail}(fibseq))) & fibseq
\end{array}$$

$$\begin{array}{lcll}
\mathsf{take}(s(a), S(\mathsf{acc}(fibseq))) & =_{\mathcal{E}} & s(\mathsf{take}(s(a), \mathsf{acc}(fibseq))) & (1)\\
& =_{\mathcal{E}} & s(\mathsf{take}(a, \mathsf{tail}(\mathsf{acc}(fibseq)))) & \mathsf{take}\\
& =_{\mathcal{E}} & s(\mathsf{take}(a, \mathsf{plus}(\mathsf{acc}(fibseq), \mathsf{tail}(fibseq)))) & \mathsf{acc}\\
& =_{\mathcal{E}} & s(+(\mathsf{take}(a, \mathsf{acc}(fibseq)), \mathsf{take}(a, \mathsf{tail}(fibseq)))) & (3)\\
& =_{\mathcal{E}} & +(s(\mathsf{take}(a, \mathsf{acc}(fibseq))), \mathsf{take}(a, \mathsf{tail}(fibseq))) & +\\
& =_{\mathcal{E}} & +(\mathsf{take}(a, S(\mathsf{acc}(fibseq))), \mathsf{take}(a, \mathsf{tail}(fibseq))) & (1)\\
& =_{\mathcal{E}} & +(\mathsf{take}(a, \mathsf{tail}(\mathsf{tail}(fibseq))), \mathsf{take}(a, \mathsf{tail}(fibseq))) & IH\\
& =_{\mathcal{E}} & \mathsf{take}(a, \mathsf{plus}(\mathsf{tail}(\mathsf{tail}(fibseq)), \mathsf{tail}(fibseq))) & (3)\\
& =_{\mathcal{E}} & \mathsf{take}(a, \mathsf{plus}(\mathsf{tail}(fibseq), \mathsf{tail}(\mathsf{tail}(fibseq)))) & (2)\\
& =_{\mathcal{E}} & \mathsf{take}(a, \mathsf{tail}(\mathsf{plus}(fibseq, \mathsf{tail}(fibseq)))) & \mathsf{plus}\\
& =_{\mathcal{E}} & \mathsf{take}(s(a), \mathsf{plus}(fibseq, \mathsf{tail}(fibseq))) & \mathsf{take}\\
& =_{\mathcal{E}} & \mathsf{take}(s(a), \mathsf{tail}(\mathsf{tail}(fibseq))) & fibseq
\end{array}$$

4.6 Two-sided infinite sequences

So far we have only considered one-sided infinite sequences, which start by head and continue infinitely into one direction. A natural extension of this are the two-sided infinite sequences, also starting by head but now continuing infinitely in two directions. An investigation of such sequences is given in [4].

For one-sided infinite sequences we used the operators $\mathsf{head} : i \to d$ and $\mathsf{tail} : i \to i$ to access elements. For two-sided infinite sequences we use a similar technique, but instead of tail we now have the two operators $L : i \to i$ and $R : i \to i$ which move the viewing position to the left or to the right respectively.

Earlier we considered one-sided infinite sequences as mappings from natural numbers to some data type. Similarly, we can view two-sided infinite sequences as mappings from integers to some data type. For the one-sided infinite sequence a we used $\mathsf{take}(n, a)$ to describe the n-th element of a, where n is a natural number. Similarly, for the two-sided infinite sequence a we can use $\mathsf{take}(n, a)$ to describe the n-th element of a, where n is now an integer rather than a natural number.

The integers can be described by the functions $0 :\to int$, $s : int \to int$, and $p : int \to int$, which is similar to how we defined the natural numbers with the additional function p where $p(x)$ is the predecessor of x.

In this situation we can do induction by choosing $C = \{0, s, p\}$. However, in this situation distinct ground terms over C may be convertible. For example, $p(s(0)) = 0$. This is interesting as in some of the older work this is not allowed, i.e. for $t, u \in \mathcal{T}(C)$ convertibility $t \sim_{\mathcal{E}} u$ only holds if t and u are the same term, so $t = u$. This restriction is not necessary for our techniques. However, we will assume that $\mathcal{E}$ always contains the equations $p(s(x)) = x$ and $s(p(x)) = x$ throughout this section.

For one-sided infinite sequences we defined $\mathsf{take} : nat, i \to b$ as follows:

$$\begin{aligned} \mathsf{take}(0, xs) &= \mathsf{head}(xs) \\ \mathsf{take}(s(x), xs) &= \mathsf{take}(x, \mathsf{tail}(xs)) \end{aligned}$$

Similarly, for two-sided infinite sequences we can now describe $\mathsf{take} : int, i \to b$ as follows:

$$\begin{aligned} \mathsf{take}(0, xs) &= \mathsf{head}(xs) \\ \mathsf{take}(s(x), xs) &= \mathsf{take}(x, R(xs)) \\ \mathsf{take}(p(x), xs) &= \mathsf{take}(x, L(xs)) \end{aligned}$$

We now define equality of two-sided infinite sequences such that two two-sided infinite sequence are equal if and only if their elements at every position are equal.

Definition 4 (Equality of two-sided infinite sequences)
Let the set $\mathcal{E}$ of equations contain the three equations $\mathsf{take}(0, xs) = \mathsf{head}(xs), \mathsf{take}(s(x), xs) = \mathsf{take}(x, R(xs))$ and $\mathsf{take}(p(x), xs) = \mathsf{take}(x, L(xs))$. Let t,u be terms of sort i satisfying $\mathsf{take}(x, t) =_{\mathcal{E}} \mathsf{take}(x, u)$.

Then $[t] = [u]$.

Now if we combine this with Theorem 2 we obtain the following Theorem for proving equality of two-sided infinite sequences:

Theorem 20 (Induction two-sided infinite sequences)
Let the set $\mathcal{E}$ of equations contain the three equations $\mathsf{take}(0, xs) = \mathsf{head}(xs), \mathsf{take}(s(x), xs) = \mathsf{take}(x, R(xs))$ and $\mathsf{take}(p(x), xs) = \mathsf{take}(x, L(xs))$. Let t,u be terms of sort i satisfying $\mathsf{take}(0, t) =_{\mathcal{E}} \mathsf{take}(0, u), \mathsf{take}(s(a), t) =_{\mathcal{E}'} \mathsf{take}(s(a), u)$, and $\mathsf{take}(p(a), t) =_{\mathcal{E}'} \mathsf{take}(p(a), u)$ where $\mathcal{E}' = \mathcal{E} \cup \{\mathsf{take}(a, t) = \mathsf{take}(a, u)\}$.

Then $[t] = [u]$.

In this setting we also obtain a double induction variant:

Theorem 21 (Double induction two-sided infinite sequences)
Let the set $\mathcal{E}$ of equations contain the three equations $\mathsf{take}(0, xs) = \mathsf{head}(xs), \mathsf{take}(s(x), xs) = \mathsf{take}(x, R(xs))$ *and* $\mathsf{take}(p(x), xs) = \mathsf{take}(x, L(xs))$. *Let t,u be terms of sort i satisfying* $\mathsf{take}(0, t) =_{\mathcal{E}} \mathsf{take}(0, u), \mathsf{take}(s(0), t) =_{\mathcal{E}} \mathsf{take}(s(0), u), \mathsf{take}(s(s(a)), t) =_{\mathcal{E}'} \mathsf{take}(s(s(a)), u)$, *and* $\mathsf{take}(p(a), t) =_{\mathcal{E}'} \mathsf{take}(p(a), u)$ *where* $\mathcal{E}' = \mathcal{E} \cup \{\mathsf{take}(a, t) = \mathsf{take}(a, u), \mathsf{take}(s(a), t) = \mathsf{take}(s(a), u)\}$.
Then $[t] = [u]$.

However, translating the strong induction variant to this setting is problematic. This would require a well-founded relation on the integers, represented by the terms in $\mathcal{T}(C)$. For the natural numbers it was fairly trivial to find such a relation, and in most cases taking the standard less-than relation was sufficient. Unfortunately this does not work for the integers. Ideally we would use the same relation, since it extends naturally to the setting with integers. Unfortunately this does not work, since the relation would not be well-founded on the integers. It might be possible to find a different well-founded relation, which would mean the theorem can be applied. However, it is not clear how useful this would be.

Example 21
Let us consider proving associativity of the operator plus *on two-sided infinite sequences. Earlier we proved that* plus *on one-sided infinite sequences is associative in Example 18, i.e.* $\mathsf{plus}(\mathsf{plus}(x, y), z) = \mathsf{plus}(x, \mathsf{plus}(y, z))$. *We will now do the same for* plus *in the setting with two-sided infinite sequences, where* plus *is now defined by:*

$$\begin{aligned} \mathsf{head}(\mathsf{plus}(x, y)) &= +(\mathsf{head}(x), \mathsf{head}(y)) \\ R(\mathsf{plus}(x, y)) &= \mathsf{plus}(R(x), R(y)) \\ L(\mathsf{plus}(x, y)) &= \mathsf{plus}(L(x), L(y)) \end{aligned}$$

Similar to before, in order to prove the goal the lemma that + is associative is required: $+(+(x, y), z) = +(x, +(y, z))$ *(1).*

$\mathsf{take}(0, \mathsf{plus}(x, \mathsf{plus}(y, z)))$	$=_{\mathcal{E}}$	$\mathsf{head}(\mathsf{plus}(x, \mathsf{plus}(y, z)))$	take
	$=_{\mathcal{E}}$	$+(\mathsf{head}(x), \mathsf{head}(\mathsf{plus}(y, z)))$	plus
	$=_{\mathcal{E}}$	$+(\mathsf{head}(x), +(\mathsf{head}(y), \mathsf{head}(z)))$	plus
	$=_{\mathcal{E}}$	$+(+(\mathsf{head}(x), \mathsf{head}(y)), \mathsf{head}(z))$	(1)
	$=_{\mathcal{E}}$	$+(\mathsf{head}(\mathsf{plus}(x, y)), \mathsf{head}(z))$	plus
	$=_{\mathcal{E}}$	$\mathsf{head}(\mathsf{plus}(\mathsf{plus}(x, y), z))$	plus
	$=_{\mathcal{E}}$	$\mathsf{take}(0, \mathsf{plus}(\mathsf{plus}(x, y), z))$	take
$\mathsf{take}(p(a), \mathsf{plus}(x, \mathsf{plus}(y, z)))$	$=_{\mathcal{E}}$	$\mathsf{take}(a, L(\mathsf{plus}(x, \mathsf{plus}(y, z))))$	take
	$=_{\mathcal{E}}$	$\mathsf{take}(a, \mathsf{plus}(L(x), L(\mathsf{plus}(y, z))))$	plus
	$=_{\mathcal{E}}$	$\mathsf{take}(a, \mathsf{plus}(L(x), \mathsf{plus}(L(y), L(z))))$	plus
	$=_{\mathcal{E}}$	$\mathsf{take}(a, \mathsf{plus}(\mathsf{plus}(L(x), L(y)), L(z)))$	*IH*
	$=_{\mathcal{E}}$	$\mathsf{take}(a, \mathsf{plus}(L(\mathsf{plus}(x, y)), L(z)))$	plus
	$=_{\mathcal{E}}$	$\mathsf{take}(a, L(\mathsf{plus}(\mathsf{plus}(x, y), z)))$	plus
	$=_{\mathcal{E}}$	$\mathsf{take}(p(a), \mathsf{plus}(\mathsf{plus}(x, y), z))$	take
$\mathsf{take}(s(a), \mathsf{plus}(x, \mathsf{plus}(y, z)))$	$=_{\mathcal{E}}$	$\mathsf{take}(a, R(\mathsf{plus}(x, \mathsf{plus}(y, z))))$	take
	$=_{\mathcal{E}}$	$\mathsf{take}(a, \mathsf{plus}(R(x), R(\mathsf{plus}(y, z))))$	plus
	$=_{\mathcal{E}}$	$\mathsf{take}(a, \mathsf{plus}(R(x), \mathsf{plus}(R(y), R(z))))$	plus
	$=_{\mathcal{E}}$	$\mathsf{take}(a, \mathsf{plus}(\mathsf{plus}(R(x), R(y)), R(z)))$	*IH*
	$=_{\mathcal{E}}$	$\mathsf{take}(a, \mathsf{plus}(R(\mathsf{plus}(x, y)), R(z)))$	plus
	$=_{\mathcal{E}}$	$\mathsf{take}(a, R(\mathsf{plus}(\mathsf{plus}(x, y), z)))$	plus
	$=_{\mathcal{E}}$	$\mathsf{take}(s(a), \mathsf{plus}(\mathsf{plus}(x, y), z))$	take

Which proves $[\mathsf{plus}(\mathsf{plus}(x, y), z)] = [\mathsf{plus}(x, \mathsf{plus}(y, z))]$ *by Theorem 20.*

Chapter 5

Implementation

We have implemented the techniques described in this document in a tool. In fact, all of the induction proofs in the examples presented in this document were automatically generated using our tool. In this chapter we will describe some details about the implementation of this tool.

5.1 Conversion

The main building block of our tool consists of seeking for a conversion of two terms given a set of equations over a set of function symbols Σ. We will describe how this is done in this section.

Finding a conversion between two terms is done using a Breadth-First Search (BFS) approach. We perform a bidirectional search, i.e. we start a BFS from both terms and try to find a point where these two searches meet. This is also known as a meet-in-the-middle approach.

Let the goal equation be $t = u$. We start the search with just the two terms t, u. Let T and U be the set of terms that t and u can be rewritten to using the equations respectively. Initially we define $T = \{t\}$ and $U = \{u\}$.

Now in each BFS step, we rewrite all terms in T and U using all equations in $\mathcal{E}$. All the newly obtained terms obtained from rewriting T are added to T, and similarly all new terms created from the terms in U are added to U.

A conversion is found between t and u if there is a term that is both convertible with t and u. In other words, t and u are convertible if and only if after some amount of steps there is a common term in T and U, meaning that the intersection of T and u must be non-empty: $T \cap U \neq \epsilon$.

Let $t \sim_{\mathcal{E}}^{\leq n} u$ describe the notion that t is convertible to u in at most n rewriting steps. Then our BFS approach searches for increasing n, a term v such that

$$t \sim_{\mathcal{E}}^{\leq n} v \sim_{\mathcal{E}}^{\leq n} u.$$

In practice we limit the BFS to a certain number of steps k. If this number of steps is reached, the conversion is considered to be failed. This does not necessarily mean that conversion is impossible, since a conversion might be found if the number of BFS steps were greater. However, performing a large number of BFS steps is typically not possible in practice. The number of terms in T and U increases in every step. This increase is often exponential in size. Therefore, after a certain number of steps it simply becomes too memory and time consuming to proceed. The method for searching conversion is described in more detail in Algorithm 1, which uses Algorithm 2 as a subroutine.

Algorithm 1 Convertible($\mathcal{E} : Set < Equation >, t : Term, u : Term, k : Integer$) : Boolean

```
1: T ← {t}
2: U ← {u}
3: for i = 0 until i = k do
4:     T ← T ∪ RewriteTerms(T, E)
5:     U ← U ∪ RewriteTerms(U, E)
6:     intersection ← T ∩ U
7:     if intersection ≠ ∅ then
8:         return True
9: return False
```

The following algorithm describes how a new set of terms is created from an initial set of terms by applying equations in $\mathcal{E}$. Here rewriting using the equations is either done from left to right or in both directions, which will be explained later.

Algorithm 2 RewriteTerms($terms : Set < Term >, \mathcal{E} : Set < Equation >$) : $Set < Term >$

```
1: result ← ∅
2: for Each term in terms do
3:     for Each equation in E do
4:         if equation can be applied on term or one of its subterms then
5:             Rewrite term to new by applying equation
6:             result ← result ∪{new}
7: return result
```

The number of terms that are generated depends on the number of equations and the shape of the equations itself. Some equations can generate a huge amount of terms. For example assume we are proving some equation over the natural numbers, and there exists an equation that says adding 0 to any natural number x results in x: $+(0, x) = x$. If we naively seek for a conversion as described above, this equation will be used on all terms of sort natural in every step. Assume we start with the term 0:

1. After 1 BFS step we have $\{0, +(0, 0)\}$.
2. After 2 BFS steps we have $\{0, +(0, 0), +(0, +(0, 0))\}$.

This is still manageable, but now assume that we also have the equation $-(0, x) = 0$. Then the following happens:

1. 0
2. $0, +(0, 0), -(0, 0)$
3. $0, +(0, 0), -(0, 0), +(0, +(0, 0)), +(0, -(0, 0)), -(0, -(0, 0)), -(0, +(0, 0))$

Now the growth at each step i is already 2^i. Often we have a lot more than just two equations of type natural, which means the amount of terms that are generated simply becomes too high after a certain number of steps. This means that the conversion search will take too long, and at some point we run out of memory.

Fortunately in a large number of cases the equations are designed such that a conversion can be found by only using the equations in one direction, from left to right. Meaning, we now search for a term v such that

$$t \to_{\mathcal{E}}^{\leq n} v \leftarrow_{\mathcal{E}}^{\leq n} u$$

for increasing n. In contrast to the earlier method this is not complete.

Our tool also supports this, by toggling a flag the equations will either only be used in one direction or in both directions. Some examples can only be proved by using the equations in both directions, for instance Example 20. An alternative option to rewriting all equations in both directions, is to only rewrite a part of the equations in both directions. This can be done by disabling rewriting in both directions, and for each equation that should be rewritten in both directions, we simply add both directions of the equation to $\mathcal{E}$. For example, if the equation $+(0, x) = x$ should be used in both directions, we include both $+(0, x) = x$ and $x = +(0, x)$ in $\mathcal{E}$.

This method is correct for any set of equations $\mathcal{E} \cup \mathcal{E}'^R$ such that $\mathcal{E}' \subseteq \mathcal{E}$ and R indicates that the equations are reversed. In our experience this method is efficient for $\mathcal{E}' = \emptyset$ or $\mathcal{E}'$ containing a small number of equations.

Furthermore the rewriting of terms by applying all equations in $\mathcal{E}$ can be done in parallel. This significantly speeds up the computation. One way to accomplish this is to divide the set of equations into a number of distinct subsets n. Then n threads are created and the sets of equations are divided over these threads, such that each thread rewrites all terms by applying only the equations in its assigned set. This is implemented in our tool.

5.2 Strategy

The main strategy we use to prove ground convergence is to simply apply Theorem 2. That is, we are given a set of functions Σ, a subset $C \subseteq \Sigma$, a set of equations $\mathcal{E}$ over these functions, and a goal $l = r$. The assumption of Theorem 2 that for all $t \in \mathcal{T}(\Sigma)_s$ there exists $u \in \mathcal{T}(C)$ such that $u \sim_{\mathcal{E}} t$ is not checked, so it is up to the user to verify that this is the case. Other tools can be helpful to prove this. For example the tool *AProVe* [10] can automatically prove termination of a TRS. In combination with Theorem 5 this can be used to prove this requirement in a lot of cases.

More specifically, the following strategy is used by our prover.

1. Try to find a conversion between the two terms directly, such that we do not apply induction unless necessary. If a conversion is found, the goal has been proven and the program exits. Otherwise, we continue by applying induction as follows.
2. Since we do not know which induction variable to use, the tool simply picks the first variable that occurs in the goal which has the same sort s as the functions in C.
3. Try induction on this variable x, for each function $f : s_1, \ldots, s_n \to s$ in C:
   (a) Generate new constants $a_1, \ldots, a_n$ of sort $s_1, \ldots, s_n$ respectively.
   (b) Create the term $f(a_1, \ldots, a_n)$.
   (c) Substitute x in the goal by this term, such that the new goal is $l[x := f(a_1, \ldots, a_n)] = r[x := f(a_1, \ldots, a_n)]$.
   (d) Generate the hypotheses H consisting of $l[x := a_i] = r[x := a_i]$ for all a_i of sort s, and create the new set of equations $\mathcal{E}' = \mathcal{E} \cup H$. In this thesis we have only considered functions in C with an arity of at most 1, hence only one hypothesis was created. An example where this is not the case are binary trees, defined by the constant $\mathsf{nil} : tree$ and the function $T : tree, tree \to tree$. Since T has an arity of 2, in step (a) we create a_1 and a_2, and hence in this step we create two hypotheses.
   (e) Try to find a conversion between $l[x := f(a_1, \ldots, a_n)]$ and $r[x := f(a_1, \ldots, a_n)]$ using the equations in $\mathcal{E}'$.
   (f) If a conversion is not found after a certain number of steps, the induction is considered to have *failed*.
4. If this succeeds for all functions in C, we have proven the goal.
5. If induction failed for one of the functions in C, we try the next variable. If induction fails on all variables, we fail to prove the goal.

Specifically for natural numbers (and integers) we use a slightly different strategy. If induction fails to prove $l[x := s(a)] = r[x := s(a)]$, we try the double induction variant. It is possible to continue in this manner and perform triple induction, and so on. In certain cases this can be useful. We do not need to do any extra work to deal with infinite objects, since these can simply be specified using equations and the take operator.

5.3 Input syntax

The input to the tool consists of a text file that should be formatted as follows. The input can be divided into three groups, function descriptions, equations, and the proof goal. These should be separated by a single empty line.

1. The first group declares the functions in Σ. Each function is declared on a separate line, and the functions are specified as follows:

   ```
   <Name> <Input sort>* <Output sort>
   ```

   For example, if we want to declare the + function on natural numbers we can do this as follows:

   ```
   + nat nat nat
   ```

   In other words: + is a function that takes two inputs of type *nat* and outputs a term of type *nat*. The functions that should be in the set of constructor $C \subseteq \Sigma$ are specified by putting $**$ in front of the function name, followed by a space. If we want to specify the natural numbers as constructors and the + function on natural numbers, this is done as follows:

   ```
   ** 0 nat
   ** s nat nat
   + nat nat nat
   ```

   It is important that each function that is used in the equations is specified in this group. Symbols that are not specified as functions will be considered as variables. Therefore, if a function is not specified but is used in the equations it will be considered a variable, which results in unintended behavior and incorrect proofs as a consequence.

2. The second group declares the equations in $\mathcal{E}$. The expected format is two terms, separated by a = sign.

   ```
   <Left term> = <Right term>
   ```

   Each term consists of functions declared in the first group, and variables which are all symbols that are not declared as functions. Functions are specified using the standard notation for functions. This means we write the function name, followed by its arguments separated by commas, all between brackets, for example $+(x, y)$. For constants we leave out the brackets, for example the function 0 is simply written as 0 rather than 0().

3. The final group consists of a single line which specifies the equation that should be proven (the goal). This is done in the same way as the other equations.

A complete example of an input file can be found in Appendix A.

5.4 Automatic lemma search

As we have seen in many of the examples in this paper, it is often necessary to introduce auxiliary equations (lemmas) to prove a goal. Our tool is often capable of generating these lemmas by itself.

With this option enabled, we use a slightly modified strategy to prove our goal. First we try to prove the goal in the way described in Section 5.2. In case this fails, we start generating auxiliary lemmas. The process of discovering and proving lemmas will be described further on. After each lemma that is found, another attempt is made at proving the goal using induction.

5.4.1 Discovering and proving lemmas

First, we create the terms $f(x_1, \ldots, x_n)$ for each $f \in \Sigma$ and fresh variables x_i for $i = 1, \ldots, n$. We add all of these terms and the variables that were created to a set. Next, for a certain number of times k, we combine these terms by replacing the variables by other terms in this set. By doing this we create a larger set of terms, with terms that have an increased depth.

After generating a set of terms in this way, we try to create equations by extracting pairs of terms from the set. If the terms in the pair have a different sort, we discard the equation. Similarly, if the two terms are already convertible using the equations that are given, we also discard the equation. In this case the lemma is not considered useful, since it follows directly from the given equations. If the equation passes these two basic tests, we try to prove it using induction. If this succeeds, the lemma is added to the set of equations $\mathcal{E}$. Otherwise, we discard the equation.

Algorithm 3 GenerateLemmas($\mathcal{E} : Set < Equation >, \Sigma : Set < Function >, C : Set < Function >, goal : Equation, k : Integer$) : Modifies $\mathcal{E}$

```
terms ← ∅
for f : s_1, ..., s_n → s ∈ Σ do
    Create fresh variables x_1, ..., x_n of sorts s_1, ..., s_n respectively.
    terms ← terms ∪{f(x_1, ..., x_n)} ∪ ⋃_{i=1}^{n} {x_i}
for i = 1 until k do
    temp ← ∅
    for term ∈ terms do
        temp ← temp ∪ CombineTerms(terms, term)
    terms ← terms ∪ temp
for t_1 ∈ terms do
    for t_2 ∈ terms do
        if t1.sort == t2.sort then
            if ¬(t_1 ∼_E t_2) then
                if induction(E, Σ, C, t_1 = t_2) then
                    E ← E ∪ {t_1 = t_2}
```

Algorithm 4 CombineTerms($terms : Set < Term >, term : Term$) : $Set < Term >$

```
result ← {term}
for each variable var in term do
    for t ∈ terms do
        if t.sort == var.sort then
            Substitute var in term by t
            result ← CombineTerms(terms, term)
return result
```

This description is supposed to explain the general ideas behind the lemma searching method. The actual algorithm differs slightly from this description due to several optimizations.

Searching for lemmas in this way has proven to be useful in many of the examples we have considered in this paper. Most of the proofs that required us to add auxiliary lemmas can now be proven fully automatically, for instance examples 14, 15, 18 and 19. In some cases the tool fails

to prove an example where multiple lemmas are required fully automatically, but the amount of lemmas that is required is significantly reduced.

An interesting observation is that the proofs found in this way are sometimes different than what we would expect. For instance, the proof of Example 19 found in this way does not use commutativity of + as a lemma, which would be the most obvious choice if we proved this goal manually. Instead, it finds associativity of + as a lemma and is able to generate a proof by addition with 0 in an interesting way:

$$
\begin{array}{lll}
\mathsf{take}(0, \mathsf{plus}(y, z)) & =_{\mathcal{E}} \; \mathsf{head}(\mathsf{plus}(y, z)) & \mathsf{take} \\
& =_{\mathcal{E}} \; +(\mathsf{head}(y), \mathsf{head}(z)) & \mathsf{plus} \\
& =_{\mathcal{E}} \; +(\mathsf{head}(y), +(0, \mathsf{head}(z))) & + \\
& =_{\mathcal{E}} \; +(+(\mathsf{head}(y), 0), \mathsf{head}(z)) & \text{Lemma: } +(x, +(y, z)) = +(+(x, y), z) \\
& =_{\mathcal{E}} \; +(0, +(\mathsf{head}(z), \mathsf{head}(y))) & \text{Lemma: } +(x, +(y, z)) = +(+(x, y), z) \\
& =_{\mathcal{E}} \; +(0, \mathsf{head}(\mathsf{plus}(z, y)) & \mathsf{plus} \\
& =_{\mathcal{E}} \; +(0, \mathsf{take}(0, \mathsf{plus}(z, y))) & \mathsf{take} \\
& =_{\mathcal{E}} \; \mathsf{take}(0, \mathsf{plus}(z, y)) & +
\end{array}
$$

5.5 Future improvements

5.5.1 Strong induction

Unfortunately strong induction has not been fully implemented in our tool. Due to its complexity it is not as suitable for automation as the standard induction variant.

The first problem is that the set $\mathcal{T}(C)$ often has an infinite size. Therefore, there can also be an infinite number of well-founded relations on $\mathcal{T}(C)$.

Secondly, assuming that we do find a suitable well-founded relation on $\mathcal{T}(C)$, we have to prove that $\varphi(v[a_1 := w_1, \ldots, a_n := w_n]) < f(w_1, \ldots, w_n)$ for $v \in \mathcal{T}(\Sigma_s \cup \bigcup_{i=1}^{n} \{a_i\})$, for all $w_1, \ldots, w_n \in \mathcal{T}(C)$ and a well-founded binary relation $<$ on $\mathcal{T}(C)$, as we did for the half function in 9. This already required quite some effort to do manually, so to do this fully automatically would likely be extremely difficult.

Despite this, we can simulate strong induction by manually adding the hypothesis to the list of equations. This allows us to solve the previous two problems manually, and still generate proofs using the strong induction variant, although it is not done fully automatically. The strong induction proofs found in this document have been generated in this way.

Chapter 6

Conclusions

In this thesis we have described how we can prove ground convertibility by induction. We started by considering finite terms and gave some examples on natural numbers. The most basic version of induction is described by Theorem 2. We observed that proving ground convertibility is undecidable. Therefore, there exists no algorithm which can decide whether two terms t, u are ground convertible given a set of equations. Next, we considered some variants of the basic version. We described a double induction variant, which roughly speaking applies induction twice.

One of the main contributions of this paper is a new variant which we call strong induction. We observed that the basic induction technique fails on certain instances where we need a stronger hypothesis. This new version of induction allowed us to prove some examples that could not be proved by the other induction variants. We mainly applied this in cases where a function h was used, where $h(x)$ represents half of x (rounded down): $\lfloor \frac{x}{2} \rfloor$. Unfortunately this new method requires a lot more effort to apply than the other methods, and is therefore not as suitable for automation.

In this thesis we were mostly interested on proving equalities on infinite terms, such as infinite sequences of some basic data type. The main method of proving such equalities on infinite objects is called *Circular Coinduction*. We showed how our induction techniques can also be applied on infinite objects, by defining a function take which takes an element from an infinite object. We defined equality on infinite objects such that two objects are equal if and only if all elements are equal. In other words, two infinite objects t, u are considered equal if and only if $\mathsf{take}(x, t) =_{\mathcal{E}} \mathsf{take}(x, u)$ for all x. By combining the induction variants with this definition of equality, we obtain methods to prove equality of two infinite objects.

We showed that the combination of the basic induction variant described in Theorem 2 and this definition of equality resulted in a technique that is equivalent to Circular Coinduction. Our initial expectation was that strong induction applied on infinite objects would be strongly related to the notion of special contexts. These are contexts satisfying a certain special property, which allows the use of additional hypotheses in Circular Coinduction. Unfortunately this does not seem to be the case and we were not able to show a relation between these two.

We applied our techniques on several non-trivial examples. We considered *morphic sequences*, which arise from the (infinite) iteration of a morphism, and infinite sequences of natural numbers. We also briefly described how our techniques can be applied on different kinds of infinite objects, like two-sided infinite sequences. In this setting distinct ground terms can be convertible, which is assumed not to be the case in most applications of inductive theorem proving.

The techniques described in this thesis were implemented in a tool. The main building block of this tool is seeking for conversion of two terms. This is done by starting a Breadth-First Search on both terms until we find a term that is convertible with both terms. Automating the induction techniques is straightforward, and is done by simply adding the hypotheses to the list of equations

and seeking for conversion of the goal.

Using this tool we were able to prove many non-trivial examples automatically. Often it is required to add auxiliary lemmas to the set of equations before the goal can be proved. Our tool is capable of discovering many of these lemmas automatically. This is accomplished by generating small terms of a limited depth. Equations are generated from these terms, and the tool attempts to prove them using induction. Once an equation (lemma) has been proved, it is added to the set of equations and another attempt at proving the original goal is made. This allows us to prove examples that previously required manual assistance fully automatically.

Bibliography

[1] Jean-Paul Allouche, Julien Cassaigne, Jeffrey Shallit, and Luca Q. Zamboni. A taxonomy of morphic sequences. *CoRR*, abs/1711.10807, 2017. 30

[2] Jean-Paul Allouche and Jeffrey O. Shallit. *Automatic Sequences. Theory, Applications, Generalizations*. 01 2003. 30, 31

[3] Raymond Aubin. Mechanizing structural induction part i: Formal system. *Theor. Comput. Sci.*, 9:329–345, 1979. 1

[4] Wieb Bosma and Hans Zantema. Ordering sequences by permutation transducers. *Indagationes Mathematicae*, 28, 12 2016. 37

[5] Robert S. Boyer and J. Strother Moore. *A Computational Logic Handbook*. Academic Press Professional, Inc., San Diego, CA, USA, 1988. 1

[6] Alan Bundy, Andrew Stevens, Frank Harmelen, Andrew Ireland, and Alan Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 12 1995. 1

[7] Koen Claessen, Moa Johansson, Dan Rosén, and Nicholas Smallbone. Automating inductive proofs using theory exploration. In Maria Paola Bonacina, editor, *Automated Deduction – CADE-24*, pages 392–406, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. 1

[8] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude - a High-performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*. Springer-Verlag, Berlin, Heidelberg, 2007. 2

[9] Lucas Dixon and Jacques D. Fleuriot. Isaplanner: A prototype proof planner in isabelle. volume 2741, pages 279–283, 07 2003. 1

[10] Jurgen Giesl, Peter Schneider-Kamp, and Stephan Falke. Aprove: A system for proving termination. 01 2003. 6, 17, 41

[11] Joseph Goguen, Kai Lin, and Grigore Rosu. Circular coinductive rewriting. *Proceedings, Automated Software Engineering '00*, 07 2000. 2

[12] Joseph A. Goguen, Kai Lin, and Grigore Roşu. Conditional circular coinductive rewriting with case analysis. In Martin Wirsing, Dirk Pattinson, and Rolf Hennicker, editors, *Recent Trends in Algebraic Development Techniques*, pages 216–232, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. 2

[13] Eugen-Ioan Goriac, Dorel Lucanu, and Grigore Roşu. Automating coinduction with case analysis. pages 220–236, 09 2010. 2

[14] Karel Hrbacek and Thomas Jech. *Introduction to Set Theory, Third Edition, Revised and Expanded*. Chapman & Hall/CRC Pure and Applied Mathematics. Taylor & Francis, 1999. 5

[15] Yaqing Jiang, Petros Papapanagiotou, and Jacques Fleuriot. Machine learning for inductive theorem proving. In Jacques Fleuriot, Dongming Wang, and Jacques Calmet, editors, *Artificial Intelligence and Symbolic Computation*, pages 87–103, Cham, 2018. Springer International Publishing. 1

[16] Yaqing Jiang, Petros Papapanagiotou, and Jacques Fleuriot. *Machine Learning for Inductive Theorem Proving: 13th International Conference, AISC 2018, Suzhou, China, September 16–19, 2018, Proceedings*, pages 87–103. 01 2018. 1

[17] Dorel Lucanu and Grigore Roşu. Circular coinduction with special contexts. In *Proceedings of the 11th International Conference on Formal Engineering Methods (ICFEM'09)*, volume 5885 of *Lecture Notes in Computer Science*, pages 639–659, 2009. 2, 21, 29

[18] Dorel Lucanu and Grigore Roşu. Circ: A circular coinductive prover. In Till Mossakowski, Ugo Montanari, and Magne Haveraaen, editors, *Algebra and Coalgebra in Computer Science*, pages 372–378, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg. 2

[19] Petros Papapanagiotou and Jacques D. Fleuriot. The boyer-moore waterfall model revisited. *CoRR*, abs/1808.03810, 2018. 1

[20] Grigore Roşu and Dorel Lucanu. Circular coinduction: A proof theoretical foundation. In Alexander Kurz, Marina Lenisa, and Andrzej Tarlecki, editors, *Algebra and Coalgebra in Computer Science*, pages 127–144, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. 2, 19, 21

[21] Hans Zantema. Turtle graphics of morphic sequences. *Fractals*, 24(01):1650009, 2016. 30

[22] Hans Zantema. Automated reasoning, 11 2018. `https://www.win.tue.nl/~hzantema/arho.pdf`. 6

[23] Hans Zantema. Equational reasoning, induction and infinite sequences. 02 2019. `https://www.win.tue.nl/~hzantema/ind.pdf`. 2, 9, 13, 17, 19, 22

[24] Hans Zantema and Jörg Endrullis. Proving equality of streams automatically. In M. Schmidt-Schlauß, editor, *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications (RTA 2011, Novi Sad, Serbia, May 30-June 1, 2011)*, LIPIcs: Leibniz International Proceedings in Informatics, pages 393–408. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2011. 2, 21, 27, 29

Appendix A

Example input file

The following is an example of input given to the tool. This example proves that $x + 0 = 0 + x$ for integers, specified by the functions 0, s(uccessor), and p(redecessor).

```
// Functions
** 0 nat
** s nat nat
** p nat nat
+ nat nat nat

// Equations
+(0,x) = x
+(s(x),y) = s(+(x,y))
+(p(x),y) = p(+(x,y))
s(p(x)) = x
p(s(x)) = x

// Goal
+(x,0) = +(0,x)
```

The corresponding output of the tool is as follows:

```
Goal: +(x, 0) = +(0, x)
Trying induction variable: x
To prove: +(0, 0) = +(0, 0)
Conversion:
+(0, 0)
To prove: +(p(a1), 0) = +(0, p(a1))
Hypotheses: +(a1, 0) = +(0, a1)
Conversion:
+(p(a1), 0) =
p(+(a1, 0)) =
p(+(0, a1)) =
p(a1) =
+(0, p(a1))
To prove: +(s(a1), 0) = +(0, s(a1))
Hypotheses: +(a1, 0) = +(0, a1)
Conversion:
+(s(a1), 0) =
s(+(a1, 0)) =
s(+(0, a1)) =
s(a1) =
+(0, s(a1))
```

Appendix B

Generated proofs

B.1 Double induction

Lemma 1 Example 10

```
Goal: h(d(x)) = x
Trying induction variable: x
To prove: h(d(0)) = 0
Conversion:
h(d(0)) =
h(0) =
0
To prove: h(d(s(a1))) = s(a1)
Added hypotheses: h(d(a1)) = a1
Conversion:
h(d(s(a1))) =
h(s(s(d(a1)))) =
s(h(d(a1))) =
s(a1)
```

Lemma 2 Example 10

```
Goal: take(x, inv(zip(inv(y), inv(y)))) = take(x, zip(y, y))
Trying induction variable: y
Skipping variable y, sort does not match sort of C
Trying induction variable: x
To prove: take(0, inv(zip(inv(y), inv(y)))) = take(0, zip(y, y))
Conversion:
take(0, inv(zip(inv(y), inv(y)))) =
head(inv(zip(inv(y), inv(y)))) =
not(head(zip(inv(y), inv(y)))) =
not(head(inv(y))) =
not(not(head(y))) =
head(y) =
head(zip(y, y)) =
take(0, zip(y, y))
To prove: take(s(a1), inv(zip(inv(y), inv(y)))) = take(s(a1), zip(y, y))
Added hypotheses: take(a1, inv(zip(inv(y), inv(y)))) = take(a1, zip(y, y))
Trying double induction
To prove: take(s(0), inv(zip(inv(y), inv(y)))) = take(s(0), zip(y, y))
Conversion:
take(s(0), inv(zip(inv(y), inv(y)))) =
take(0, tail(inv(zip(inv(y), inv(y))))) =
take(0, inv(tail(zip(inv(y), inv(y))))) =
head(inv(tail(zip(inv(y), inv(y))))) =
not(head(tail(zip(inv(y), inv(y))))) =
not(head(zip(inv(y), tail(inv(y))))) =
not(head(inv(y))) =
not(not(head(y))) =
```

```
head(y) =
head(zip(y, tail(y))) =
head(tail(zip(y, y))) =
take(0, tail(zip(y, y))) =
take(s(0), zip(y, y))
To prove: take(s(s(a1)), inv(zip(inv(y), inv(y)))) = take(s(s(a1)), zip(y, y))
Added hypotheses: take(s(a1), inv(zip(inv(y), inv(y)))) = take(s(a1), zip(y, y))
Conversion:
take(s(s(a1)), inv(zip(inv(y), inv(y)))) =
take(s(a1), tail(inv(zip(inv(y), inv(y))))) =
take(s(a1), inv(tail(zip(inv(y), inv(y))))) =
take(s(a1), inv(zip(inv(y), tail(inv(y))))) =
take(a1, tail(inv(zip(inv(y), tail(inv(y)))))) =
take(a1, inv(tail(zip(inv(y), tail(inv(y)))))) =
take(a1, inv(zip(tail(inv(y)), tail(inv(y))))) =
take(a1, inv(zip(inv(tail(y)), inv(tail(y))))) =
take(a1, zip(tail(y), tail(y))) =
take(a1, tail(zip(y, tail(y)))) =
take(a1, tail(tail(zip(y, y)))) =
take(s(a1), tail(zip(y, y))) =
take(s(s(a1)), zip(y, y))
```

B.2 Morphic sequences

Lemma Example 14

```
Goal: take(x, zip(y, inv(y))) = take(x, f(y))
Trying induction variable: y
Skipping variable y, sort does not match sort of C
Trying induction variable: x
To prove: take(0, zip(y, inv(y))) = take(0, f(y))
Conversion:
take(0, zip(y, inv(y))) =
head(zip(y, inv(y))) =
head(y) =
head(f(y)) =
take(0, f(y))
To prove: take(s(a1), zip(y, inv(y))) = take(s(a1), f(y))
Added hypotheses: take(a1, zip(y, inv(y))) = take(a1, f(y))
Trying double induction
To prove: take(s(0), zip(y, inv(y))) = take(s(0), f(y))
Conversion:
take(s(0), zip(y, inv(y))) =
take(0, tail(zip(y, inv(y)))) =
take(0, zip(inv(y), tail(y))) =
head(zip(inv(y), tail(y))) =
head(inv(y)) =
not(head(y)) =
head(tail(f(y))) =
take(0, tail(f(y))) =
take(s(0), f(y))
To prove: take(s(s(a1)), zip(y, inv(y))) = take(s(s(a1)), f(y))
Added hypotheses: take(s(a1), zip(y, inv(y))) = take(s(a1), f(y))
Conversion:
take(s(s(a1)), zip(y, inv(y))) =
take(s(a1), tail(zip(y, inv(y)))) =
take(a1, tail(tail(zip(y, inv(y))))) =
take(a1, tail(zip(inv(y), tail(y)))) =
take(a1, zip(tail(y), tail(inv(y)))) =
take(a1, zip(tail(y), inv(tail(y)))) =
take(a1, f(tail(y))) =
take(a1, tail(tail(f(y)))) =
take(s(a1), tail(f(y))) =
take(s(s(a1)), f(y))
```

Lemma Example 15

```
Goal: take(x, even(zip(y, z))) = take(x, y)
Trying induction variable: y
Skipping variable y, sort does not match sort of C
Trying induction variable: z
Skipping variable z, sort does not match sort of C
Trying induction variable: x
To prove: take(0, even(zip(y, z))) = take(0, y)
Conversion:
take(0, even(zip(y, z))) =
head(even(zip(y, z))) =
head(zip(y, z)) =
head(y) =
take(0, y)
To prove: take(s(a1), even(zip(y, z))) = take(s(a1), y)
Added hypotheses: take(a1, even(zip(y, z))) = take(a1, y)
Conversion:
take(s(a1), even(zip(y, z))) =
take(a1, tail(even(zip(y, z)))) =
take(a1, even(tail(tail(zip(y, z))))) =
take(a1, even(tail(zip(z, tail(y))))) =
take(a1, even(zip(tail(y), tail(z)))) =
take(a1, tail(y)) =
take(s(a1), y)
```

Lemma 1 Example 16

```
Goal: take(x, diff(zip(y, inv(y)))) = take(x, zip(zeroes, inv(diff(y))))
Trying induction variable: y
Skipping variable y, sort does not match sort of C
Trying induction variable: x
To prove: take(0, diff(zip(y, inv(y)))) = take(0, zip(zeroes, inv(diff(y))))
Conversion:
take(0, diff(zip(y, inv(y)))) =
head(diff(zip(y, inv(y)))) =
and(head(zip(y, inv(y))), head(tail(zip(y, inv(y))))) =
and(head(zip(y, inv(y))), head(zip(inv(y), tail(y)))) =
and(head(zip(y, inv(y))), head(inv(y))) =
and(head(zip(y, inv(y))), not(head(y))) =
and(head(y), not(head(y))) =
0 =
head(zeroes) =
head(zip(zeroes, inv(diff(y)))) =
take(0, zip(zeroes, inv(diff(y))))
To prove: take(s(a1), diff(zip(y, inv(y)))) = take(s(a1), zip(zeroes, inv(diff(y))))
Added hypotheses: take(a1, diff(zip(y, inv(y)))) = take(a1, zip(zeroes, inv(diff(y))))
Trying double induction
To prove: take(s(0), diff(zip(y, inv(y)))) = take(s(0), zip(zeroes, inv(diff(y))))
Conversion:
take(s(0), diff(zip(y, inv(y)))) =
take(0, tail(diff(zip(y, inv(y))))) =
head(tail(diff(zip(y, inv(y))))) =
head(diff(tail(zip(y, inv(y))))) =
and(head(tail(zip(y, inv(y)))), head(tail(tail(zip(y, inv(y)))))) =
and(head(zip(inv(y), tail(y))), head(tail(zip(inv(y), tail(y))))) =
and(head(inv(y)), head(tail(zip(inv(y), tail(y))))) =
and(not(head(y)), head(tail(zip(inv(y), tail(y))))) =
and(not(head(y)), head(zip(tail(y), tail(inv(y))))) =
and(not(head(y)), head(tail(y))) =
not(and(head(y), head(tail(y)))) =
not(head(diff(y))) =
head(inv(diff(y))) =
head(zip(inv(diff(y)), tail(zeroes))) =
take(0, zip(inv(diff(y)), tail(zeroes))) =
take(0, tail(zip(zeroes, inv(diff(y))))) =
take(s(0), zip(zeroes, inv(diff(y))))
```

```
To prove: take(s(s(a1)), diff(zip(y, inv(y)))) = take(s(s(a1)), zip(zeroes, inv(diff(y))
    ))
Added hypotheses: take(s(a1), diff(zip(y, inv(y)))) = take(s(a1), zip(zeroes, inv(diff(y
    ))))
Conversion:
take(s(s(a1)), diff(zip(y, inv(y)))) =
take(s(a1), tail(diff(zip(y, inv(y))))) =
take(a1, tail(tail(diff(zip(y, inv(y)))))) =
take(a1, tail(diff(tail(zip(y, inv(y)))))) =
take(a1, tail(diff(zip(inv(y), tail(y))))) =
take(a1, diff(tail(zip(inv(y), tail(y))))) =
take(a1, diff(zip(tail(y), tail(inv(y))))) =
take(a1, diff(zip(tail(y), inv(tail(y))))) =
take(a1, zip(zeroes, inv(diff(tail(y))))) =
take(a1, zip(zeroes, inv(tail(diff(y))))) =
take(a1, zip(tail(zeroes), inv(tail(diff(y))))) =
take(a1, zip(tail(zeroes), tail(inv(diff(y))))) =
take(a1, tail(zip(inv(diff(y)), tail(zeroes)))) =
take(a1, tail(tail(zip(zeroes, inv(diff(y)))))) =
take(s(a1), tail(zip(zeroes, inv(diff(y))))) =
take(s(s(a1)), zip(zeroes, inv(diff(y))))
```

Lemma 2 Example 16

```
Goal: take(x, f(y)) = take(x, zip(zeroes, inv(y)))
Trying induction variable: y
Skipping variable y, sort does not match sort of C
Trying induction variable: x
To prove: take(0, f(y)) = take(0, zip(zeroes, inv(y)))
Conversion:
take(0, f(y)) =
head(f(y)) =
0 =
head(zeroes) =
head(zip(zeroes, inv(y))) =
take(0, zip(zeroes, inv(y)))
To prove: take(s(a1), f(y)) = take(s(a1), zip(zeroes, inv(y)))
Added hypotheses: take(a1, f(y)) = take(a1, zip(zeroes, inv(y)))
Trying double induction
To prove: take(s(0), f(y)) = take(s(0), zip(zeroes, inv(y)))
Conversion:
take(s(0), f(y)) =
take(0, tail(f(y))) =
head(tail(f(y))) =
not(head(y)) =
head(inv(y)) =
head(zip(inv(y), tail(zeroes))) =
head(tail(zip(zeroes, inv(y)))) =
take(0, tail(zip(zeroes, inv(y)))) =
take(s(0), zip(zeroes, inv(y)))
To prove: take(s(s(a1)), f(y)) = take(s(s(a1)), zip(zeroes, inv(y)))
Added hypotheses: take(s(a1), f(y)) = take(s(a1), zip(zeroes, inv(y)))
Conversion:
take(s(s(a1)), f(y)) =
take(s(a1), tail(f(y))) =
take(a1, tail(tail(f(y)))) =
take(a1, f(tail(y))) =
take(a1, zip(zeroes, inv(tail(y)))) =
take(a1, zip(zeroes, tail(inv(y)))) =
take(a1, tail(zip(inv(y), zeroes))) =
take(s(a1), zip(inv(y), zeroes)) =
take(s(a1), zip(inv(y), tail(zeroes))) =
take(s(a1), tail(zip(zeroes, inv(y)))) =
take(s(s(a1)), zip(zeroes, inv(y)))
```

B.3 Natural numbers

Lemma example 18

```
Goal: +(x, +(y, z)) = +(+(x, y), z)
Trying induction variable: z
To prove: +(x, +(y, 0)) = +(+(x, y), 0)
Failed to prove +(x, +(y, 0)) = +(+(x, y), 0) induction on z failed.
Trying induction variable: y
To prove: +(x, +(0, z)) = +(+(x, 0), z)
Failed to prove +(x, +(0, z)) = +(+(x, 0), z) induction on y failed.
Trying induction variable: x
To prove: +(0, +(y, z)) = +(+(0, y), z)
Conversion:
+(0, +(y, z)) =
+(y, z) =
+(+(0, y), z)
To prove: +(s(a1), +(y, z)) = +(+(s(a1), y), z)
Added hypotheses: +(a1, +(y, z)) = +(+(a1, y), z)
Conversion:
+(s(a1), +(y, z)) =
s(+(a1, +(y, z))) =
s(+(+(a1, y), z)) =
+(s(+(a1, y)), z) =
+(+(s(a1), y), z)
```