Part I

Main Report

CHAPTER 1

RESEARCH REPORT

Towards automating proofs in the context of commutative semi-rings

Iannetta Paul[†]

[†]École Normale Supérieure de Lyon

August 26, 2016

Abstract

Rewriting induction (Reddy 1990) is a rather well-known method to automate the proof of inductive theorems. This report aims at describing some experiments and their results. Those experiments aimed at extending the power of rewriting induction (firstly, by changing the expansion scheme and secondly, by allowing variables to be extended in some cases). In a second part, the possibility to apply rewriting induction to well-ordering commutative semi-rings is discussed. The results obtained so far look promising enough to encourage further investigation in the same track.

1 Introduction

In papers [CAT05] and [CAT06], Chiba et al. expose a method to transform programs into other, more efficient, programs. The basic idea is to check whether a program fits a certain pattern

so that an optimisation can be proposed. However, in order to apply such a transformation the original program should satisfy some properties. For example, a function used in the original program should be associative or commutative. My first intent was to automatically prove that those required conditions were satisfied.

Since my goal was to automatize proof I was redirected to a comparative presentation [小外 00] of rewriting induction [Red90] and inductionless induction [Lan81; Com94]. Because prerequisites for rewriting induction seemed easier to satisfy I started to investigate on this topic, as of today, I do not consider inductionless induction as a priority anymore.

As I investigated rewriting induction, I looked for already known extensions, in particular, I based my work on this reformulation by Aoto [Tak08] and implemented my own engine to perform rewriting induction. After some failed experiments to prove automatically the associativity of the multiplication function without some clever but counter intuitive lemma, I tried to prove some arithmetic properties on numbers. Which leads me, belatedly, to investigate whether the model used for automated proofs could not be extended to commutative semi-rings.

This paper is organized in the following way. The first part presents term rewriting, therefore you can safely skip this section if you are already familiar with the terminology and definitions. The second part presents rewriting induction, how it works and my contribution to it as well as a description of the tool I developed to automatically handle rewriting induction. The last sections deals with proving properties in commutative semi-rings.

2 Preliminaries

Thorough all this section, the sets $\mathcal{F}$ and $\mathcal{V}$ will be supposed to be disjoint. According to their names, $\mathcal{F}$ is supposed to contain function symbols, whereas $\mathcal{V}$ contains countably many variable symbols.

2. 1 Term rewriting

This subsection contains basic definition about term rewriting. Its only purpose is to serve as a reference where all concepts used in this report are defined.

In order to show the usefulness of the following definitions. This part gradually presents the formal definition of the theorem that states the associativity of the plus operator. In order to do so, the notion of number and mathematical expression should be formalized.

Definition 1 (*Arity*) Each function $f \in \mathcal{F}$ can formally take arguments. The number of arguments a function symbol can handle is called *arity*. As a matter of convenience, the function $\mathrm{ar} : \mathcal{F} \mapsto \mathbb{N}$ which maps each function symbols to its arity will be used.

Remark 1
Conveniently, $\mathcal{F}_k$ will denote the subset of $\mathcal{F}$ whose elements are of arity k. Specifically, elements in $\mathcal{F}_0$ are called *constants*.

Definition 2 (*Signature*) The tuple $(\mathcal{F}, \mathrm{ar})$ is called signature and will be referred by $\mathcal{F}$ alone when there is no ambiguity. As a commodity, $\mathcal{F}_k$ refers to the subset of $\mathcal{F}$ containing the function of arity k.

As of now, the plus operator as well as the two operators in charge of describing PEANO numbers can be defined.

Example 1 ***Arithmetical operators***
The following signature describes a basic framework to handle the addition of number. (The arity of operators is indicated in the exponent.)

$$\mathcal{A} = \{0^0, s^1, +^2\}$$

Definition 3 (*Terms*) The notion of *term*, which standardize the notion of expression, is defined with the following recursive definition. The set of *terms* $\mathcal{T}(\mathcal{F}, \mathcal{V})$ is the least closed set under the following rules:

i) *constants* are terms ;

ii) *variables* are terms ;

iii) $\forall k \in \mathbb{N}^*, f \in \mathcal{F}_k, (t_i)_{1 \leq i \leq k} \in \mathcal{T}(\mathcal{F}, \mathcal{V}), f(t_1, \ldots, t_k)$ is a term.

Remark 2
Due to their recursive definition terms can be viewed as *tree*. Therefore, the vocabulary of tree may be used to describe terms

Remark 3
The set of terms on $\mathcal{A}$ is exactly the set of arithmetical expressions with natural numbers and the plus operator.

From now on, let us introduce some definitions to handle terms.

Definition 4 (*Position*) Let $t = f(t_1, \cdots, t_n)$ a term. t is denoted by $t_{|\epsilon}$ and t_i by $t_{|i}$. Consequently, inner positions can be defined as (when it does make sense) $t_{|i.j} = (t_{|i})_{|j}$ where $i.j$ means i concatenated to j.

Definition 5 (*Ground term*) A ground term is a term with no variables. The set of ground terms is denoted by $\mathcal{T}(\mathcal{F})$.

Definition 6 (*Substitution*) A *substitution* σ is a partial function from variables to terms. By abuse of notation, the *homomorphically extended substitution* $\hat{\sigma}$ from terms to terms will be also called substitution on denoted σ.

Definition 7 (*Domain of substitution*) The *domain* $\mathcal{D}\text{om}(\sigma)$ of substitution σ is the set $\{x \in \mathcal{V} \mid x\sigma \neq x\}$.

Remark 4
The application of a substitution σ to a term t, $\sigma(t)$ will be expressed with postfix notation $t\sigma$.

Definition 8 (*Context*) A *context* is a term with a hole. This hole might be later filled by another term. Formally, the hole ($\Box$) is added to the set of variables. Moreover, the special variable "hole" ($\Box$) should appear exactly once. Context are usually denoted by $\mathcal{C}$, a context in which the hole has been replace by the term t is denoted $\mathcal{C}[t]$.

Example 2
Consider the following context $\mathcal{C} \equiv \Box + \mathrm{s}(0)$, then $\mathcal{C}[0] \equiv 0 + \mathrm{s}(0)$

Definition 9 (*Term Rewriting System*) A *Term Rewriting System* on the set of terms is a set of term tuples (ℓ, r) called *rewrite rule*. Each term rewriting $\mathcal{R}$ system induces a rewriting relation $\to_{\mathcal{R}}$ made of the tuples $(\mathcal{C}[\ell\sigma], \mathcal{C}[r\sigma])$ where $(l, r) \in \mathcal{R}$, $\mathcal{C}$ is a context, and σ is a substitution over the set of terms with respect to $\mathcal{R}$.

Example 3
The following term rewriting ($\mathcal{A}dd$) system over $\mathcal{A}$ encodes the definition of addition.

$$(\mathcal{A}dd) : \begin{cases} 0 & + & y & \longrightarrow_{\mathcal{A}dd} & y \\ \mathrm{s}(x) & + & y & \longrightarrow_{\mathcal{A}dd} & \mathrm{s}(x+y) \end{cases}$$

Definition 10 (*Normal form*) Let $\mathcal{R}$ be a rewrite system and $\to_{\mathcal{R}}$ be the rewrite relation it induces. A term s is said to be in normal form if with respect to $\to_{\mathcal{R}}$ there does not exists some t such that $s \to_{\mathcal{R}} t$.

Definition 11 (*Weak Normalization*) A term rewriting system is weakly normalizing if all term have at least one normal form.

Definition 12 (*Termination*) A term rewriting system is normalizing if all rewriting sequences can eventually reach a term in normal form.

Definition 13 (*Defined symbols and Constructor symbols*) Given a term rewriting system $\mathcal{R}$, the set of *defined symbols* is the set of function symbols that appears as root symbol in the first element of the tuples in $\mathcal{R}$; all other symbols are *constructor symbols*.

Remark 5
Terms made up only of *defined symbols* are called defined terms and terms made up only of *constructor symbols* are called constructor terms.

Example 4
The defined symbols of $\mathcal{A}dd$ is $+$ and its constructor symbols are s and 0.

Definition 14 (*Basic terms*) A *basic term* is a term whose root symbols is a defined symbol and whose sub-terms are all constructor terms. The set of all basic terms is referred to as $\mathcal{B}$. The set of all basic subterms of a term s is denoted as $\mathcal{B}(s)$.

Definition 15 (*Sufficiently complete*) A term rewriting system is said to be *sufficiently complete* if all ground basic term can be reduced to ground constructor terms.

Considering to terms t, and s, *unification* is the process of finding a substitution σ such that $t\sigma = s\sigma$.

Definition 16 (*Unifiers*) Let $\{s_i \approx t_i\}_i$ be a set of equalities. The set $\mathcal{U}$ of *unifiers* is the set of substitutions σ such that $\forall i, s_i\sigma \approx t_i\sigma$.

Definition 17 (*Most General Unifier*) Let $\mathcal{U}$ be the set of unifier with respect to some set of equalities $\mathcal{S}$. A substitution $\sigma \in \mathcal{U}$ is said to be more general that $\sigma' \in \mathcal{U}$ if there exists a substitution τ (which may not be an unifier) such that $\sigma' = \tau\sigma$. The most general unifier will be noted as $\text{mgu}(\mathcal{S})$.

2. 2 The CafeOBJ rewriting engine

CafeOBJ is an algebraic programming language based on order-sorted equational logic, more precisely it uses term rewriting as its foundation. This language is mainly designed to formalize the specifications of software systems. Then, it is possible to use the rewriting properties of CafeOBJ to prove that the software meets its specification.

In the following parts only the rewrite engine of CafeOBJ will be used. This rewrite engine tries to apply in a non-deterministic way rewrite rules on the goal we want to prove. The reason it is used here is that it is possible to assign precedence to infix symbols which is handful when dealing with arithmetic statements.

Example 5 ***Proof of Vajda's equality***
The following proof score is the proof score describing that Vajda's equality can be proven by rewriting once the induction hypotheses are correctly found. At first, this example is just to present what CafeOBJ looks like, but, in a second reading, it is also the proof that certify that if rewriting can find the two induction hypothesis then the proof can be completed.

```
mod! M {
  [A B Bl] .
  op _+_ : B B -> B {assoc comm} .
  op _*_ : B B -> B {assoc} .
  op s : A -> A .
  op 0 : -> A .
  op f : A -> B .
  ops 2n 3n : -> A .

  op tr : -> Bl .
  op _===_ : B B -> Bl .

  vars X Y Z : B .

  eq (X + Y) * Z = (X * Z) + (Y * Z) .
  eq X * (Y + Z) = (X * Y) + (X * Z) .
  eq f(s(s(N:A))) = f(s(N)) + f(N) .
```

```
  eq (X:B === X:B) = tr .
}

open M .
op l : -> B .
eq l = (f(s(s(s(3n)))) * f(s(s(s(2n))))) + (f(3n) * f(s(2n)))
+ (f(s(s(s(s(s(s(3n))))))) * f(s(s(s(s(2n)))))) .
op r : -> B .
eq r = (f(s(s(s(s(3n))))) * f(s(s(2n))))
+ (f(s(3n)) * f(2n)) + (f(s(s(s(s(s(s(3n))))))) * f(s(s(s(s(s(2n))))))) .
red (l === r) .
close .
```

Note: The linebreaks between `eq` and `.` should not be present in the section between `open M` and `close .`

The first part (the lines starting with `op`) declares the signature (the operators, their arity and their properties); the second part describe the term rewriting system used (here are used the Fibonacci recursive definition, the distributivity rules and the standard equality); the last between `open M` and `close` is the proof score. First `l` and `r` are declared, those 'variables' refer respectively to the left and right hand-side obtained after applying the induction hypothesis in the proof of Vajda's equality.

3 Inductive theorem proving

3. 1 Presentation

Let us come back to the associativity of the plus operator. It is expressed as follows (variables are implicitly universally quantified):

$$(x + y) + z = x + (y + z)$$

In order to state the correctness of this statement, it is natural to harness the inductive definition of natural numbers (ie. a number is either zero or the successor of another number)

Therefore, basically, two proof schemes are at our disposal: case analysis and induction. For that very reason, it would be great if those two schemes could be automated to some extent and that is what rewriting induction [Red90] provides. Due to the nature of this scheme it can only handle *inductive theorems* and does not exactly perform case analysis even if in some cases the proof log might look alike to traditional case analysis.

Definition 18 (*Inductive theorems*) A property $P(t_1, \ldots, t_n)$ on terms is an *inductive theorem* if $P(t_1\sigma, \ldots, t_n\sigma)$ holds for all *ground* substitutions wherein all variables in $t_1, \ldots, t_n$ are instantiated to ground terms.

The associative property of plus as stated above is an inductive theorems as it can be prove separately on each kind of ground terms. Here, there are two such kinds: zero and numbers different from zero.

3. 2 Rewriting induction

Rewriting induction, first exposed by Reddy in 1990, is a proof technique used to handle induction theorems automatically. It offers a new approach, less cumbersome than structural induction because it does not require meta-variables; furthermore there is no need to take quantifiers into account since they are implicit and have no role in the algorithm.

Its dark side is that it still cannot compete against the power of structural induction as it will be shown later. The variant presented here follows [Tak08], in the aforementioned Aoto presents as well an extension to deal with non-orientable equalities (which will not be used in the following).

Informally, the idea behind rewriting induction is to start with to set of equations: one containing the theorems to be proved, the other containing hypothesis that can be used to complete the proof. Once all equalities have been cleared the hypothesis are proved.

Let us now present it in full details.

First, since rewriting induction needs an order on terms, let's introduce the following order, *lexicographic path order*:

Definition 19 (*Lexicographic path order*) Let Σ be a signature and $\succ$ an order on Σ. The lexicographic path order $\succ_{lpo}$ on $\mathcal{T}(\mathcal{F}, \mathcal{V})$ induced by $s \succ_{lpo} t$ is defined as follows:

LPO 1 : t is a variable in s and $t \neq s$, or

LPO 2 : $s = f(s_1, \ldots, s_m), t = g(t_1, \ldots, t_n)$, and

LPO 2a : $\exists i, 1 \leq i \leq m$ with $s_i \succ_{lpo} t$, or

LPO 2b : $f \succ g$ and $s \succ_{lpo} t_j, \forall j, 1 \leq j \leq n$, or

LPO 2c : $f = g, s \succ_{lpo} t_j, \forall j, 1 \leq j \leq n$ and $(s_1, \ldots, s_m) \succ_{lpo}^{lex} (t_1, \ldots, t_n)$.

Definition 20 (*Equality with respect to a term rewriting system*) Let $\mathcal{R}$ be a term rewriting system and $\rightarrow_{\mathcal{R}}$ the relation induced by $\mathcal{R}$. Two terms s and t are equal (denoted $s \approx t$) with respect to $\mathcal{R}$ if and only if, for all substitutions σ, $s\sigma \leftrightarrow_{\mathcal{R}}^{*} r\sigma$ where $\leftrightarrow_{\mathcal{R}}^{*}$ denotes the reflexive symmetric transitive closure of $\rightarrow_{\mathcal{R}}$.

The inference rules describing rewriting induction can be found in Figure 1.1. Those rules are to be read from bottom to top, the bottom is transformed in the form exposed above the bar. The rules `simplify` and `delete` removes superfluous equalities while the `expand` rule is the keystone of this scheme. $(E, H) \rightsquigarrow_{\alpha} (E', H')$ means that rule α transforms (E, H) to (E', H').

Theorem 1 (*Rewriting induction*) For a *sufficiently complete* and *terminating* TRS $\mathcal{R}$, if $\langle \{s = t\}, \emptyset \rangle \rightsquigarrow_{e,s,d}^{*} \langle \emptyset, H \rangle$ then $\mathcal{R} \vdash s = t$.

3. 3 Strengths and weaknesses

The main strength of this approach is that since variables are implicitly (universally) quantified, there is no need to jungle with quantifiers applying the rules. Another point worth noting is that the inference rules remain quite basic, in the sense that implementing them in a functional language is not so much of a daunting task.

Expand:

$$\frac{\langle E \cup \mathrm{Expd}_u(s,t), H \cup \{s \to t\}\rangle}{\langle E \cup \{s = t\},\ H\rangle}, s_{|u} \in \mathcal{B}, s > t$$

Simplify:

$$\frac{\langle E \cup \{s' = t\}, H\rangle}{\langle E \cup \{s = t\}, H\rangle},\ s \to_{R \cup H} s'$$

Delete:

$$\frac{\langle E, H\rangle}{\langle E \cup \{s = s\}, H\rangle}$$

$$\mathrm{Expd}_u(s,t) = \{C[r]\sigma = t\sigma \mid s \equiv C[s_{|u}],\ \sigma = \mathrm{mgu}(s_{|u}, l),\ l \to r \in \mathcal{R}\}$$

Figure 1.1: Rewriting induction inductive rules

Example 6 ***Proof of associativity of plus***
In order to prove that the inductive theorem about the associativity of the plus operator holds in the rewriting system $\mathcal{A}dd$, let's input the equality relative to associativity into the rewriting induction framework and see how it unravels on that theorem.

$$\begin{aligned}
&\langle\{x + (y + z) = (x + y) + z\}, \emptyset\rangle \\
\rightsquigarrow_e &\langle\{x + (y + 0) = (x + y); x + (y + s(z')) = s((x + y) + z)\}, \\
&\{x + (y + z) \to (x + y) + z\}\rangle \\
\rightsquigarrow_s^* &\langle\{x + y = (x + y); x + s(y + z') = s((x + y) + z')\}, \\
&\{x + (y + z) \to (x + y) + z\}\rangle \\
\rightsquigarrow_s^* &\langle\{x + y = (x + y); s(x + (y + z')) = s((x + y) + z')\}, \\
&\{x + (y + z) \to (x + y) + z\}\rangle \\
\rightsquigarrow_d^* &\langle\emptyset, \{x + (y + z) \to (x + y) + z\}\rangle
\end{aligned}$$

By the means of the expand rules an induction hypothesis is generated, then using this hypothesis as well as the rules of the rewriting system $\mathcal{A}dd$ the equalities set can be reduced to the empty set. Therefore, the theorem about the associativity of the plus operator holds.

But this apparent simplicity in design brings in weaknesses as well. The main weakness is the very strong dependency on the rewriting system used. This dependency will often cause the expand rule to create unusable hypothesis. Moreover it will often cause the system to diverge.

This weakness is clearly illustrated in an attempt to prove the associativity of the max operator.

Example 7 ***Fail in the proof of the associativity of max***
The rewriting system for the max operator is as follows:

$$(\mathcal{M}ax) : \begin{cases} \max(0, y) & \longrightarrow_{\mathcal{M}ax} \quad y \\ \max(x, 0) & \longrightarrow_{\mathcal{M}ax} \quad x \\ \max(\mathrm{s}(x), \mathrm{s}(y)) & \longrightarrow_{\mathcal{M}ax} \quad \mathrm{s}(\max(x, y)) \end{cases}$$

The theorem stating the associativity of the max operator is as follows:

$$\max(\max(x, y), y) = \max(x, \max(y, z))$$

Let's apply rewriting induction:

$$\begin{aligned}
&\langle\{\max(\max(x, y), z) = \max(x, \max(y, z))\}, \emptyset\rangle \\
\rightsquigarrow_e &\langle\{\max(\max(0, y), z)) = max(0, z),\ \max(\max(x, 0), z)) = max(0, z) \\
&\max(\max(\mathrm{s}(x), \mathrm{s}(y)), z) = \max(\mathrm{s}(x), \max(s(y), z))\}, \\
&\{\max(\max(x, y), y) \to \max(x, \max(y, z))\}\rangle \\
\rightsquigarrow^*_{s,d} &\langle\{\max(\mathrm{s}(\max(x, y)), z) = \max(\mathrm{s}(x), \max(\mathrm{s}(y), z))\}, \\
&\max(\max(x, y), y) \to \max(x, \max(y, z))\}\rangle
\end{aligned}$$

It is still possible to expand here but it is just the beginning of an infinite loop.
In that case, it is possible to prove this theorem by structural induction. Moreover, it is possible to extend rewriting induction to handle this case, but even if it should work theoretically it is not suited for automation.

In this case, the system winds up in an infinite loop because it fails to expand the right part. More precisely, since the expansion is not deterministic it may not take the good path. Another point to consider is that rewriting induction will never try to expand variables since they are not considered as basic terms.

3. 4 Expansion scheme revisited

In this subsection, two tracks to extend rewriting induction will be explored: the first one is a new approach in the expansion process; the second explores what can be done if variables become expandable.

The first extension to be discussed is parallel extensions, instead of expending at one position at a time all expendable position are expanded. If the previous approach was to be compared with a tree traversal it would be a depth-first search, with this new approach it becomes a breadth-first search. The idea behind this variation is that it can prevent the expansion process to run blindly into an infinite loop (like in the case of the proof of the associativity of max.

Theorem 2 (*Parralel Expansion*) Parallel expansion is equivalent to normal extensions.

Sadly, it is still not enough to proof the associativity of the max operator. So as to do that, here comes the second extension: variable extension. The idea here is to remove the condition that the term s should be basic in the expansion rule and to allow variables as well. The striking problem of this approach is that the number of positions can be performed increases dramatically. Thus, in practice, variables are expanded if and only the expand rules can be applied nowhere else.

Theoretically, this extension solves the proof of the max operator. (It suffices to expand the z variable. Nevertheless, in practice, since there is always a basic term to expand, this extension is never triggered and the proof that max is associative fails again.

Theorem 3 (*Variable expansion*) Variable expansion preserves equality with respect to the base term rewriting system.

Example 8 ***Associativity of*** max ***revisited***
Let's assume that variables can be expanded. Then, continuing from the last state in the previous example:

$$\begin{aligned}
&\langle \{\max(\mathrm{s}(\max(x,y)),z) = \max(\mathrm{s}(x),\max(\mathrm{s}(y),z))\},\\
&\quad \{\max(\max(x,y),y) \to \max(x,\max(y,z))\}\rangle\\
\rightsquigarrow_e &\langle \{\max(\mathrm{s}(\max(x,y)),\max(0,y_1) = \max(\mathrm{s}(x),\max(\mathrm{s}(y),0)),\\
&\quad \max(\mathrm{s}(\max(x,y)),\max(x_1,0) = \max(\mathrm{s}(x),\max(\mathrm{s}(y),0)),\\
&\quad \max(\mathrm{s}(\max(x,y)),\max(\mathrm{s}(x_1),\mathrm{s}(x_2)) = \max(\mathrm{s}(x),\max(\mathrm{s}(y),\mathrm{s}(\max(x_1,y_1))),\\
&\quad \{\max(\max(x,y),y) \to \max(x,\max(y,z))\}\rangle\\
\rightsquigarrow^*_{d,s} &\langle \emptyset, \{\max(\max(x,y),y) \to \max(x,\max(y,z))\}
\end{aligned}$$

In that case, it is possible to conclude the validity of the associativity of the max operator.

3. 5 Automation of Rewriting Induction

In order to ease experiments and to take full advantage of the fact that rewriting induction is to be automatized, a program in haskell has been written. This program allows to perform rewriting induction and implements some of the extension presented in this report. More precisely, the program can apply the expand rule in parallel (variables expansion is not explicitly supported), it is also possible to give a set of goals to prove and to give a partial order so as to help the program to choose which goals should be proven first. The order used to compare terms is the lexicographic path order. This order has been chosen for its simplicity (implementation-wise).

More details about the implementation can be found in the section 1

4 Proving properties in well ordered semi rings

In this section, well-ordered semi-rings as well as how to harness their properties to improve rewriting induction in this particular setting will be discussed.

4. 1 A special case of Vajda's equality

The theorem at hand is extracted from standard arithmetic and is more precisely about Fibonacci numbers. The theorem known as *Vajda's equality* is as follow:

$$\forall (i,j,n) \in \mathbb{N}^3, \mathcal{F}_{n+i}\mathcal{F}_{n+j} - \mathcal{F}_n\mathcal{F}_{n+i+j} = (-1)^n\mathcal{F}_i\mathcal{F}_j$$

Nevertheless, in order to simplify the problem, the variables i, j and n are respectively set to m, 1 and $2m$ where m is an integer. This leads to the following theorem:

$$\forall n \in \mathbb{N}, P(n)$$

where

$$P(n) : \mathcal{F}_{3n}\mathcal{F}_{2n+1} - \mathcal{F}_{2n}\mathcal{F}_{3n+1} = \mathcal{F}_n$$

In this section, the state of the research will be exposed. The result are still not complete.

Assuming that $P(n)$ and $P(n+1)$ are valid for a specific n, a traditional second order induction would yield the following induction hypothesis:

$$\begin{aligned} P(n) &: \mathcal{F}_{3n}\mathcal{F}_{2n+1} - \mathcal{F}_{2n}\mathcal{F}_{3n+1} &&= \mathcal{F}_n \\ P(n+1) &: \mathcal{F}_{3n+3}\mathcal{F}_{2n+3} - \mathcal{F}_{2n+2}\mathcal{F}_{3n+4} &&= \mathcal{F}_{n+1} \end{aligned}$$

Again, according to the traditional inductive scheme,

$$P(n+2) : \mathcal{F}_{3n+6}\mathcal{F}_{2n+5} - \mathcal{F}_{2n+4}\mathcal{F}_{3n+7} = \mathcal{F}_{n+2}$$

is to be proved.

Since the final objective is to automatize it using only natural numbers, the terms in the above equations should be rearranged so that the minus sign is no longer used. Furthermore, $\mathcal{F}_{n+2}$ should be expanded with respect to the definition and the induction hypothesis should be applied. Which leads to,

$$\mathcal{F}_{3n+6}\mathcal{F}_{2n+5} + \mathcal{F}_{2n+2}\mathcal{F}_{3n+4} + \mathcal{F}_{2n}\mathcal{F}_{3n+1} = \mathcal{F}_{2n+4}\mathcal{F}_{3n+7} + \mathcal{F}_{3n}\mathcal{F}_{2n+1} + \mathcal{F}_{3n+3}\mathcal{F}_{2n+3}$$

Here, the properties coming from the well-ordered semi-ring structure come in handy. Let's consider the multiple of n ($2n$ and $3n$ as special not expandable constant terms like 0). From there on, by using commutativity, associativity and the fact term can terms can internally be sorted, deciding equality becomes as easy as deciding the equality of two ordered sets.

Nevertheless, as of now, the rewriting induction framework does not support second order induction (this point will be further discussed in the Extension section). Thus, the process is still not fully automated. To make things worse, rearranging terms according in the equalities so as to get rid of minus symbols create rewrite rules that are difficult to use.

4. 2 Associativity and commutativity

As seen above, associativity and commutativity are the keystones behind the success of the previous attempt. (Even though, the attempt is still not completely automatized it still looks promising). Furthermore, those properties weight even more than the real definition of the plus and times operator used in the proof of Vajda's equality.

Theorem 4 () In a well ordered semi-ring checking equality is the same as checking the equality of two ordered sets.

4. 3 Orderering the ring elements

Since the ring has to be well-ordered it has to be equipped with a well-founded order. The order chosen is of little importance since it does not matter even it is choosen randomly. But performance-wise the choice does matter, in particular, the more trival it is to decide whether an element is greater than another the better.

In the case of the proof of the Vajda's equality, the only kind of terms appearing are products of two Fibonnaci numbers, therefore, ordering them with Lexicographic order make sense. More precisely, a product $\mathcal{F}_n\mathcal{F}_m$ is seen as a tuple (n, m) and since the productthe case of the proof of the Vajda's equality, the only kind of terms appearing are products of two Fibonnaci numbers, therefore, ordering them with Lexicographic order make sense. More precisely, a product $\mathcal{F}_n\mathcal{F}_m$ is seen as a tuple (n, m) and since the product is commutative n is always chosen so that it is greater or equal to m.

5 Extension

This section describes future works and what could be done to improve the work described in this report.

5. 1 Handling second order induction

Theoretically speaking, handling second order induction is as simple as handling traditional induction. Indeed, to reduce second order induction to first order induction it suffices to consider the property $Q(n) : P(n) \wedge P(n + 1)$.

Nonetheless, since rewriting induction is extremely dependent on the form of rewriting rules, it might prove more difficult to automate in a satisfying way.

5. 2 Working with equalities

As long as commutativity holds it is possible to handle equalities as expressions that could be handled as terms. Commutativity is important because it does not matter whether there are minus signs appearing in the left or right side of equalities since they can be swapped to the other side easily.

5. 3 Get rid of commutativity

As of now, commutativity is central to achieve proof in the semi-ring settings, allowing us to move around terms quite freely. Nevertheless, this is a strong assumption and it would be great if it could be removed. That being said, it does not seem obvious at all to get rid of it and moreover it may require a whole different approach.

6 Conclusion

This paper proposed some extensions to rewriting induction, parallel expansions and variable expansions. Those extension increases slightly the power of rewriting induction by limiting the risk that the extension path proves to be infinite.

On the other hand, this paper discussed the possibility to apply rewriting induction to well-ordered semi-rings through the example of a special case of Vajda's identity. Nevertheless, the work is still mostly in progress because even if it has been certified by CafeOBJ that the proof can be automated, it has not been proven in the strict setting of rewriting induction.

7 Acknowledgements

I want to thank warmly my supervisors at JAIST (Dr. Hirokawa Nao and Dr. Chiba Yuki) for their valuable advice all along my research period. I also want to thank the other member of my research team, Jungo Shibuya, Netrakom Park and Watanabe Ryouko who all contributed to the very positive atmosphere of our laboratory. Last but not least, I want to thank the Rhône-Alpes region (now Auvergne-Rhône-Alpes) and the École Normale Supérieure de Lyon for their grant that allowed me to spend a year studying in Japan.

BIBLIOGRAPHY

[ALM15] Martin Avanzini, Ugo Dal Lago, and Georg Moser. "Analysing the complexity of functional programs: higher-order meets first-order". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*. 2015, pp. 152–164. DOI: 10.1145/2784731.2784753. URL: http://doi.acm.org/10.1145/2784731.2784753.

[CAT05] Yuki Chiba, Takahito Aoto, and Yoshihito Toyama. "Program transformation by templates based on term rewriting". In: *Proceedings of the 7th ACM-SIGPLAN International Conference on Principles and Practice of Declarative Programming (PPDP 2005)*. ACM Press, 2005, pp. 59–69.

[CAT06] Yuki Chiba, Takahito Aoto, and Yoshihito Toyama. "Program transformation by templates: A rewriting framework". In: *IPSJ Transactions on Programming* 47.SIG 16 (PRO 31) (2006), pp. 52–65.

[Com94] Hubert Comon. *Inductionless Induction*. 1994.

[HMZ13] Nao Hirokawa, Aart Middeldorp, and Harald Zankl. "Uncurrying for Termination and Complexity". In: *J. Autom. Reasoning* 50.3 (2013), pp. 279–315. URL: http://dx.doi.org/10.1007/s10817-012-9248-3.

[Lan81] D.S. Lankford. *A simple explanation of inductionless induction*. Tech. rep. Mathematics Department, Louisiana Tech. Univ., 1981.

[Red90] *Term Rewriting Induction*. 1990.

[Sch98] Renate A. Schmidt. "*E*-Unification for Subsystems of *S4*". In: *Rewriting Techniques and Applications, 9th International Conference, RTA-98, Tsukuba, Japan, March 30 - April 1, 1998, Proceedings*. 1998, pp. 106–120. DOI: 10.1007/BFb0052364. URL: http://dx.doi.org/10.1007/BFb0052364.

[Tak08] Aoto Takahito. *Dealing with Non-orientable Equations in Rewriting*. 2008.

[小外 00] 小池広高 and 外山芳人. 潜在帰納法と書き換え帰納法の比較. 2000.

CHAPTER 2

RESEARCH ACTIVITIES

This chapter will briefly describes the activities I was involved in this year. In particular, I had the opportunity to join a weekly seminar organized by my research laboratory. The theme revolve mainly around term rewriting and I was a speaker roughly once a month from February 2016. Each seminar I joined is briefly presented and I mention the one where I was a speaker. Those seminars were often used to discuss our research progress and where held in a pedagogic purpose. In particular, Dr. Hirokawa's talks mainly aimed at presenting topics (related with term rewriting) that were unknown or still not really understood by the other students of our research laboratory.

Here follows the list of those seminars.

Jan 26 (Kento Yamazaki): Stringification for Termination Analysis .

Feb 9 (Paul Iannetta) Inductive Theorem Proving Based on Rewriting : This was my first talk as a speaker. I introduced to my research group a comparison between rewriting induction and inductionless induction based on the paper [小外 00]. I also justified why I choose to work on rewriting induction rather than induction-less induction: the main reason being that rewriting induction does not require confluence.

Feb 16 (Nao Hirokawa) Knuth-Bendix Order Dr. Hirokawa presented us the Knuth-Bendix order and some its applications. The purpose of this talk was mainly educational, I mean that, at the end of the talk we should be able to understand how Knuth-Bendix order works so that we can use it in our research projects if needs be.

March 22 (Park Netrakom) Associative unification In this talk Mr. Netrakom presented the concept of unification modulo associativity. That is, the associativity of operators is taken into account when trying to find an unifier.

March 29 (The Curry-Howard Isomorphism) I presented to my group the Curry-Howard Isomorphism. In this talk I tried to do a parallelism between types in functional programming language and first-order logic.

Apr 12 (Nao Hirokawa) Uncurrying for termination In this talk, Dr. Hirokawa presents applicative term rewriting systems, how those system can be uncurried and then how to prove that they terminate. This talk is based on the following two papers [HMZ13; ALM15].

Apr 26 (Park Netrakom) Correctness of Unification In this talk Mr. Netrakom presented us the unification algorithm and justified its correctness. This talk was lecture-like and its purpose was to be an exercise so that the speaker had to impersonate a lecturer.

Apr 26 (Paul Iannetta) Rewriting induction in real-life In this talk, I present rewriting induction in more details and highlighted the fact that the success of rewriting induction relies heavily on the input term rewriting system.

May 10 (Jungo Shibuya) Product Construction and Subset Construction Mr. Shibuya presented in a lecture-like talk (as Mr. Netrakom did two weeks before) how to construct the cartesian product of two automata and how to transform a non deterministic automaton to a deterministic automaton. He also explained why the constructed automata were correct.

May 10 (Ryouko Watanabe) Proving Normalization by Persistency Decomposition Ms. Watanabe presented the concept of persitency decomposition (which is roughly decompose the rules of a term rewriting system according to the type of the function they encode) and how this decomposition can be used to prove normalization.

May 19 (Nao Hirokawa) Abstract Completion

May 26 (Paul Iannetta) On Local Confluence and Critical Pairs I exposed in a lecture-like talk the relation between critical pairs, local confluence as well as the Newmann's lemma.

Jun 2 (Ryouko Watanabe) From Outermost Termination to Innermost Termination In this talk, Ms. Watanabe exposed an algorithm to transform an outermost terminating term rewriting system into an innermost terminating rewriting system. The main theorem of the talk was that if the newly created term rewriting system is innermost most terminating then the original term rewriting system is outermost terminating.

Jun 9 (Nao Hirokawa) Programming Turing Machines Dr. Hirokawa presented Turing machines for a term rewriting point of view.

Jun 16 (Jungo Shibuya) CKY algorithm Mr. Shibuya presented the CKY algorithm which is used to test whether a word is recognized by a context free grammar.

Jun 16 (Park Netrakom) A1-Unification Algorithm This talk was mainly based on [Sch98].

Jun 22 (Ryouko Watanabe and Paul Iannetta) This seminar was a collaboration with another research group Ms. Watanabe and Mr. Iannetta presented their respective works and the progress they made from their previous talk (see Jun 2 and May 26).

Jun 30 (Jungo Shibuya) Linear Sentences

Jul 7 (Ryouko Watanabe) Automated Normalization Analysis for Term Rewriting In this talk, Ms. Watanabe presents Nort, her automated tools to prove normalization. She also explained the different methods used by her tool.

Jul 14 (Nao Hirokawa) Confluence of monadic string rewriting system This talk address the following problem solved by Otto (JCSS 1987): given a monadic string rewriting system $\mathcal{R}$ and a string w, is R confluent on the R-equivalent class of w.

Jul 21 (Paul Iannetta) In this talk, I presented my last progresses and the application of rewriting induction to well-ordered semi-rings.

Jul 28 (Park Netrakom) Freezing Mr. Netrakom presented how to freeze rewrite rules. This freezing technique is used to improve the efficiency of tools use to automatically decide termination. It is based on [HMZ13].

Contents

I Main Report **1**

1 Research report **2**

1 Introduction . . . 2
2 Preliminaries . . . 3
2. 1 Term rewriting . . . 3
2. 2 The CafeOBJ rewriting engine . . . 6
3 Inductive theorem proving . . . 7
3. 1 Presentation . . . 7
3. 2 Rewriting induction . . . 8
3. 3 Strengths and weaknesses . . . 8
3. 4 Expansion scheme revisited . . . 10
3. 5 Automation of Rewriting Induction . . . 11
4 Proving properties in well ordered semi rings . . . 11
4. 1 A special case of Vajda's equality . . . 11
4. 2 Associativity and commutativity . . . 12
4. 3 Orderering the ring elements . . . 12
5 Extension . . . 13
5. 1 Handling second order induction . . . 13
5. 2 Working with equalities . . . 13
5. 3 Get rid of commutativity . . . 13
6 Conclusion . . . 13
7 Acknowledgements . . . 14

2 Research activities **16**

II Appendixes **20**

1 Implementation details . . . 21
2 More details on the syntax . . . 21

Part II

Appendixes

1 Implementation details

The program mentioned in the section 3. 5 will be explained with more details in this section.

The source code is divided into four parts: a front-end (lexer and parser), a part for handling operations on terms, a part that handle rewriting induction and a part that processes the dependence graph to handle multi-goals proofs.

Language choice: The Haskell language has been chosen because the functional paradigm shares a lot with term rewriting. That said I could have chosen OCaml which is a more familiar language to me, but I chose Haskell for two reasons: it was the occasion to learn a new language and the Parsec library is embedded directly in the language whereas tools like OCamllex or Menhir are external tools.

What is actually implemented The program features simple rewriting induction with parallel extension as well as basic term rewriting. The term rewriting part can handle most standard operations on terms and can reduce them with respect to a term rewriting system. For that purpose, standard unification has been implemented as well. Nevertheless all the features are not accessible or meant to be accessible to the user, at least as of now.

As of now, the program can either prove a property or fail its proof but it cannot disprove a property.

Example Here is an example describing quickly how to use the program.

Example 9 ***Associativity of the plus operator*** `def plus : Nat Nat -> Nat .`

```
var X : Nat .
var Y : Nat .

ax plus(z, Y) = Y .
ax plus(s(X), Y) = s(plus(X, Y)) .

goals plus(X, plus(Y, Z)) = plus(plus(X, Y), Z) .
```

The syntax is as follows, `def` introduces the type of element in the signature (of the subsequent term rewriting system); `ax` introduces the rewriting system and `goals` introduces a series of theorem to be proved with respect to the rewriting system.
Once this file is submitted to the program it will try to find an order on terms, if an order can be found the rules given will be oriented with respect to this order and a rewriting system will be derived; from there on rewriting induction will be applied to the goals.
In this case the program successfully complete the proof as expected.

2 More details on the syntax

The syntax of the language used is extremely regular: every command starts by the name of the command to be invoked and ends by a period.

Variables

Variables' name are in uppercase.

Terms

Terms are constructed with the following grammar:
<term> := <identifiers> (<arglist>) <arglist> := (<term> | <varname>) <arglist´> <arglist´> := e | ; <arglist>

Sort

Sorts are of the following form:

1. Basic sort name (Basic sort name should be all lowercase save for the first which has to be in uppercase) ;
2. → Basic sort name (represent a function with no parameters whose return type is Basic sort name) ;
3. ⟨sort⟩ → ⟨sort⟩

The command `def`

Syntax: def ⟨identifier⟩ : ⟨sort⟩ .
Syntax: def ⟨varname⟩ : ⟨sort⟩ .
Identifiers have to be in lowercase. Sort names are defined on the fly and does not mean much on their own but they are important for asserting type constraints that are later to be checked during type verification.

The command `ax`

Syntax: ax ⟨term⟩ : ⟨term⟩ .
This command introduces the underlying rewriting system in which all the proofs are to be done.

The command `goals`

Syntax goals [⟨goalNb⟩] [⟨goalsNb⟩] ⟨term⟩ = ⟨term⟩ (; ⟨term⟩ = ⟨term⟩ (; ...)) .
If goals are assigned numbers then they can be later referred in later goals as dependencies. The goals are then processed in topological order.