A New Implementation of Multi-Context Algebraic Inductive Theorem Prover

ChengCheng Ji, Haruhiko Sato and Masahito Kurihara

***Abstract*—Term rewriting induction (RI) is a principle for automatic inductive theorem proving proposed by Reddy. There are several strategic issues in RI: (1) which reduction order should be applied, (2) which (axiomatic or hypothetical) rules should be applied during rewriting, and (3) which variables should be instantiated for induction. The multi-context rewriting induction with termination checker (MRIt) solved these problems by creating virtual parallel processes dynamically to handle the nondeterministic choices. In this paper, we present a multi-context algebraic inductive theorem prover called *lz-itp* and implement it in a functional, object-oriented programming language Scala which features the lazy evaluation mechanism. Based on MRIt, our implementation exploits the lazy evaluation schemas to gain efficiency. Also the automatic lemma generations are employed to support solving the lemma-required problems. The experiments show that *lz-itp* is more efficient than the original MRIt implementation of Sato and Kurihara.**

***Index Terms*—Term rewriting system, Term rewriting induction, Multi-completion, Lazy evaluation.**

I. Introduction

An algebraic inductive theorem is a proposition for algebraic specifications defined on inductively-defined data structures such as natural numbers and lists. The proof of such inductive theorems plays a fundamental role in the field of formal verification of information systems. There is a method called *term rewriting induction* (RI) proposed by Reddy [4], which is a automatable proof principle for proving inductive theorems on term rewriting systems. The RI method relies on the termination of the given term rewriting systems representing the axioms, because if we have a terminating term rewriting system (i.e., there exists no infinite rewrite sequence), we can use the transitive closure of the corresponding rewrite relation of the system as a well-founded order over terms for the basis of induction.

However, there are several kinds of strategic issues in constructing successful proofs by RI:

- which reduction order should be applied
- which (axiomatic or hypothetical) rules should be applied during rewriting
- which variables should be instantiated for induction

It is not a trivial task to choose appropriate strategies in general, because of the nondeterminism during the induction procedure. Since inappropriate ones can easily lead the procedure to divergence (i.e., infinite computation), we cannot physically create and run a number of parallel processes because such naive parallelization would cause serious inefficiency. This makes it really hard to fully automate the RI-based inductive theorem proving.

ChengCheng Ji, Haruhiko Sato and Masahito Kurihara are with the Division of Computer Science and Information Technology in Graduate School of Information Science and Technology, Hokkaido University, Sappro, Japan. 060-0814. e-mail: kisyousei@complex.ist.hokudai.ac.jp, haru@complex.ist.hokudai.ac.jp, kurihara@ist.hokudai.ac.jp.

Aoto [5] proposed a variant of RI, called the rewriting induction with termination checker (RIt), which partially solved the problem by using an external automated termination checker instead of a specific reduction order. Therefore, the users no longer had to provide promising reduction orders and they could implicitly exploit modern termination proving methods more powerful than the simply parameterized reduction orders (e.g., recursive path orders and polynomial orders). However, a new issue came out: in which direction the hypothetical equations should be oriented. Since the use of the termination checker increased the possibility of success in the orientation and we could decide the direction of the equations dynamically, more flexibility in the orientation strategy was given, from the viewpoint of strategy. Based on a multi-context strategy a procedure called *multi-context rewriting induction with termination checker* (MRIt) has been proposed by Sato [7] in order to exploit such flexibility in orientation and solve the other strategic issues of RI.

In this paper, we present a new implementation of multi-context algebraic inductive theorem prover *lz-itp* based on MRIt, which efficiently simulates the execution of parallel RIt processes in a single process by dynamically dealing with the nondeterministic choices supporting automatic lemma exploration. Because MRIt relies on the manipulation of the *node* database, we exploit the lazy evaluation schemas [9] [10] to gain more efficiency. In our implementation, we also combine automatic lemma exploring techniques [12] to solve the lemma-required problems.

This paper is organized as follows. In Section II we will provide a brief review on term rewriting systems and inductive theorem proving procedure RI and MRIt. In Section III, we will discuss the implementation of *lz-itp*. The result of the experiments will be shown and discussed in Section IV. In Section V, we will conclude with possible future work.

II. Preliminaries

A. Term Rewriting Systems

Let us briefly review the basic notions for term rewriting systems(TRS) [1] [2] [3] [14] [16]. We start with the basic definitions.

Definition 2.1: A $signature$ Σ is a set of *function symbols*, where each $f \in \Sigma$ is associated with a non-negative integer n, the *arity* of f. The elements of Σ with arity n=0 are called $constant\ symbols$.

Let V be a set of $variables$ such that $\Sigma \cap V = \emptyset$. With Σ and V we can build $terms$.

Definition 2.2: The set $T(\Sigma, V)$ of all terms over Σ and V is recursively defined as follows: $V \subseteq T(\Sigma, V)$ (i.e., all variables are terms) and if $t_1, \ldots, t_n \in T(\Sigma, V)$ and $f \in \Sigma$, then $f(t_1, \ldots, t_n) \in T(\Sigma, V)$, where n is the arity of f.

For example, if f is a function symbol with arity 2 and $\{x, y\}$ are variables, then $f(x, y)$ is a term. We write $s \equiv t$

ISBN: 978-988-19253-6-7
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

when the terms s and t are identical. A term s is a $subterm$ of t, if either $s \equiv t$ or $t \equiv f(t_1, \ldots, t_n)$ and s is a $subterm$ of some $t_k (1 \leq k \leq n)$.

Variables can be replaced by terms with specified substitutions. A $substitution$ is a function $\sigma : V \to T(\Sigma, V)$ such that $\sigma(x) \neq x$ for only finitely many xs. We can extend any substitution σ to a mapping $\sigma : T(\Sigma, V) \to T(\Sigma, V)$ by defining $\sigma(f(s_1, \ldots, s_n)) = f(\sigma(s_1), \ldots, \sigma(s_n))$. The application $\sigma(s)$ of σ to s is often written as $s\sigma$. A term t is an instance of a term s if there exists a substitution σ such that $s\sigma \equiv t$. Two terms s and t are $variants$ of each other and denoted by $s \doteq t$, if s is an instance of t and vice versa (i.e., s and t are syntactically the same up to renaming variables). Now we can define TRS as follows:

Definition 2.3: A rewrite rule $l \to r$ is an ordered pair of terms such that l is not a variable and every variable contained in r is also in l. A *term rewriting system (TRS)*, denoted by R, is a set of rewrite rules.

Let $\Box$ be a new symbol which does not occur in $\Sigma \cup V$. A $context$, denoted by C, is a term $t \in T(\Sigma, V \cup \{\Box\})$ with exactly one occurrence of $\Box$. $C[s]$ denotes the term obtained by replacing $\Box$ in C with s.

Definition 2.4: The $reduction\ relation \to_R \subseteq T(\Sigma, V) \times T(\Sigma, V)$ is defined by $s \to_R t$ iff there exists a rule $l \to r \in R$, a context C, and a substitution σ such that $s \equiv C[l\sigma]$ and $C[r\sigma] \equiv t$. In particular, the relation $\leftrightarrow_R^*$ is the reflexive, symmetric, transitive closure of the rewrite relation $\to_R$. A term s is $reducible$ if $s \to_R t$ for some t; otherwise, s is a $normal\ form$.

A TRS R $terminates$ if there is no infinite rewrite sequence $s_0 \to_R s_1 \to_R \cdots$. We also say that R has the $termination$ property or R is $terminating$. The termination property of TRS can be proved by the following definition and theorem.

Definition 2.5: A strict partial order $\succ$ on $T(\Sigma, V)$ is called a $reduction\ order$ if it possesses the following properties.

- $closed\ under\ substitution$:
  $s \succ t$ implies $s\sigma \succ t\sigma$ for any substitution σ.
- $closed\ under\ context$:
  $s \succ t$ implies $C[s] \succ C[t]$ for any context C.
- $well\text{-}founded$:
  there exist no infinite decreasing sequences $t_1 \succ t_2 \succ t_3 \succ \cdots$.

Theorem 2.6: A term rewriting system R terminates iff there exists a reduction order $\succ$ that satisfies $l \succ r$ for all $l \to r \in R$.

The *root symbol* of a term $s \equiv f(s_1, \ldots, s_n)$ is f, denoted by *root(s)*. The set of all *defined symbols* of R is defined as $D_R = \{root(l) \mid l \to r \in R\}$. The set of all *constructor symbols* of R is defined as $C_R = \Sigma \backslash D_R$. A term consisting of only constructor symbols and variables is a *constructor term*.

Before we talk about *term rewriting induction*, we also make a review of the basic notions.

A term is a *basic term* if its root symbol is a defined symbol and its arguments are constructor terms. We denote all basic subterms of a term t by $\mathcal{B}(t)$. A TRS R is *ground-reducible* if every ground basic term is reducible in R. An equation $s = t$ is an *inductive theorem* of R if all its ground instances $s\sigma = t\sigma$ are equational consequences of the equational axioms R (regarded as a set of equations), i.e.,$s\sigma \leftrightarrow_R^* t\sigma$.

B. Term Rewriting Induction

The term rewriting induction (RI) is an automatable proof principle for proving inductive consequences of equational axioms proposed by Reddy[4]. Given a set $\mathcal{R}$ of rewrite rules and a reduction order $\succ$ containing $\mathcal{R}$, RI works on a pair of a set of equations $\mathcal{E}$ and a set of rewrite rules $\mathcal{H}$. Intuitively, $\mathcal{E}$ represents conjectures (i.e., theorems and lemmas) to be proved and $\mathcal{H}$ represents inductive hypotheses applicable to $\mathcal{E}$.

The inference rule of RI is defined as follows:

DELETE: $\langle \mathcal{E} \cup \{s = s\}, \mathcal{H} \rangle \vdash \langle \mathcal{E}, \mathcal{H} \rangle$

SIMPLIFY: $\langle \mathcal{E} \cup \{s = t\}, \mathcal{H} \rangle \vdash \langle \mathcal{E} \cup \{s' = t\}, \mathcal{H} \rangle$
if $s \to_{\mathcal{R} \cup \mathcal{H}} s'$

EXPAND: $\langle \mathcal{E} \cup \{s = t\}, \mathcal{H} \rangle \vdash$
$\langle \mathcal{E} \cup \mathrm{Expd}_u(s, t), \mathcal{H}\{s \to t\} \rangle$
if $u \in \mathcal{B}(s)$ and $s \succ t$

where the function $\mathrm{Expd}_u(s, t)$ is defined as the following:

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

Let $s = t$ be an equation such that it can be oriented from s to t as a rewrite rule $s \to t$. Given such an equation $s = t$ and a basic subterm u of s, $\mathrm{Expd}_u(s, t)$ returns a set of equations by overlapping u with the basic left-hand sides l of rewrite rules $l \to r$ of $\mathcal{R}$. The resultant equations will be used as new conjectures in the EXPAND inference rule for a case analysis to cover the original conjecture $s = t$ if $\mathcal{R}$ is ground-reducible. The DELETE rule simply removes trivial equations. The SIMPLIFY rule simplify the equations using a rule of $\mathcal{R}$ and $\mathcal{H}$.

Given a set of equations $\mathcal{E}_0$, a ground-reducible terminating TRS $\mathcal{R}$, and a reduction order $\succ$ containing $\mathcal{R}$, if there is a derivation sequence $\langle \mathcal{E}_0, \mathcal{H}_0 \rangle \vdash_{RI} \langle \mathcal{E}_1, \mathcal{H}_1 \rangle \vdash_{RI} \cdots \vdash_{RI} \langle \mathcal{E}_n, \mathcal{H}_n \rangle$ where $\mathcal{H}_0 = \mathcal{E}_n = \emptyset$, then all equations in $\mathcal{E}_0$ are inductive theorems of $\mathcal{R}$.

The choice of the reduction order $\succ$ is important for the success of inductive theorem proving with the rewriting induction. However, it is often not easy to provide a suitable reduction order before the procedure starts nor to choose appropriate inference rules to be applied in the reasoning steps. This problem is partially solved by a new system called RIt proposed by [5]. The RIt is a variant of the term rewriting induction, using an arbitrary termination checker instead of a reduction order by modifying the EXPAND rule to:

EXPAND: $\langle \mathcal{E} \cup \{s = t\}, \mathcal{H} \rangle \vdash$
$\langle \mathcal{E} \cup \mathrm{Expd}_u(s, t), \mathcal{H}\{s \to t\} \rangle$
if $u \in \mathcal{B}(s)$ and
$\mathcal{R} \cup \mathcal{H} \cup \{s \to t\}$ terminates

It allows us to use more powerful termination checking techniques. However, it becomes more important to choose an appropriate direction for an equation to be expanded, since we can often orient the equation in both directions. Moreover, the strategy using in simplification also plays an important role in term rewriting induction[7].

ISBN: 978-988-19253-6-7
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

C. Multi-context Rewriting Induction

Trying to pursue different choices of context of RI, the multi-context rewriting induction procedure with termination checkers (MRIt) simulates execution of multiple RIt processes based on the framework of MKB [7] [8]. MRIt works on a set of nodes, where the node is defined as follows:

Definition 2.7: A *node* is a tuple $\langle s:t, H_1, H_2, E\rangle$, where $s:t$ is an ordered pair of terms s and t called $datum$, and H_1, H_2, E are subsets of I called $labels$ such that:

- H_1, H_2 and E are mutually disjoint. (i.e., $H_1 \cap H_2 = H_1 \cap E = H_2 \cap E = \emptyset$)
- $i \in H_1$ implies $s \succ_i t$, and $i \in H_2$ implies $t \succ_i s$

Intuitively, E represents all processes containing $s = t$ as a conjecture to be proved, and H_1 (resp. H_2) represents all processes containing $s \to t$ (resp.$t \to s$) as an inductive hypothesis.

Note that the process index in MRIt is a sequence of natural numbers. Unlike MKB, the set of possible indices$I$ is infinite in MRIt because the number of running processes is not fixed: the procedure starts with one root process ϵ and in the course of the execution, new processes would be created by forking existing processes when there appear nondeterministic choices in applying rules.

Given the current set N of nodes, $(\mathcal{E}[N,i];\mathcal{H}[N,i])$ defined in the following represents the current set of conjectures and hypothesis in a process p_i.

Definition 2.8: Let $n = \langle s:t, H_1, H_2, E\rangle$ be a node and $i \in I$ be an index. The $\mathcal{E}$*-projection* $\mathcal{E}[n,i]$ of n onto i is a (singleton or empty) set of conjectures defined by

$$\mathcal{E}[n,i] = \begin{cases} \{s \leftrightarrow t\}, & \text{if } i \in E, \\ \emptyset, & \text{otherwise.} \end{cases}$$

Similarly, the $\mathcal{H}$*-projection* $\mathcal{H}[n,i]$ of n onto i is a set of hypothesis defined by

$$\mathcal{H}[n,i] = \begin{cases} \{s \to t\}, & \text{if } i \in H_1, \\ \{t \to s\}, & \text{if } i \in H_2, \\ \emptyset, & \text{otherwise.} \end{cases}$$

These notions can also be extended for a set N of nodes as follows:

$$\mathcal{E}[N,i] = \bigcup_{n\in N} \mathcal{E}[n,i], \qquad \mathcal{H}[N,i] = \bigcup_{n\in N} \mathcal{H}[n,i]$$

The inference rules of MRIt are defined as following :

DELETE: $N \cup \{\langle s:s, H_1, H_2, E\rangle\} \vdash N$

EXPAND: $N \cup \{\langle s:t, H_1, H_2, E \cup E'\rangle\} \vdash$
$N \cup \{\langle s:t, H_1 \cup E', H_2, E\rangle\} \cup$
$\{\langle s':t', \emptyset, \emptyset, E'\rangle \mid s' = t' \in \text{Expd}_u(s,t)\}$
if $E' \neq \emptyset, u \in \mathcal{B}(s)$ and $\mathcal{H}[N,i] \cup \mathcal{R} \cup \{s \to t\}$ terminates for all $i \in E'$

SIMPLIFY_R $N \cup \{\langle s:t, H_1, H_2, E\rangle\} \vdash$
$N \cup \left\{ \begin{array}{l} \langle s:t, H_1, H_2, \emptyset\rangle \\ \langle s':t, \emptyset, \emptyset, E\rangle \end{array} \right\}$
if $E \neq \emptyset$ and $s \to_{\mathcal{R}} s'$

SIMPLIFY_H: $N \cup \{\langle s:t, H_1, H_2, E\rangle\} \vdash$
$N \cup \left\{ \begin{array}{l} \langle s:t, H_1, H_2, E\backslash H\rangle \\ \langle s':t, \emptyset, \emptyset, E \cap H\rangle \end{array} \right\}$
if $E \cap H \neq \emptyset, \langle l:r, H, \ldots, \ldots\rangle \in N$, and $s \to_{\{l\to r\}} s'$

FORK: $N \vdash \psi_P(N)$
for some fork function ψ and a set P of processes in N

GC: $N \cup \{\langle s:t, \emptyset, \emptyset, \emptyset\rangle\} \vdash N$

SUBSUME: $N \cup \left\{ \begin{array}{l} \langle s:t, H_1, H_2, E\rangle \\ \langle s':t', H_1', H_2', E'\rangle \end{array} \right\} \vdash$
$N \cup \{\langle s:t, H_1 \cup H_1', H_2 \cup H_2', E''\rangle\}$
if $s:t$ and $s':t'$ are variants and
$E'' = (E\backslash(H_1' \cup H_2')) \cup (E'\backslash(H_1 \cup H_2))$

SUBSUME_P: $N \vdash sub(N,L)$
if $\forall p \in L, \exists p' \in I(N)\backslash L :$
$(\mathcal{E}[N,p], \mathcal{H}[N,p]) = (\mathcal{E}[N,p'], \mathcal{H}[N,p'])$

MRIt starts with the initial set N_0 of nodes:

$$N_0 = \{\langle s:t, \emptyset, \emptyset, \{\epsilon\}\rangle \mid s \leftrightarrow t \in \mathcal{E}_0\},$$

which means, given the initial set of conjectures $\mathcal{E}_0$ and a ground-reducible terminating TRS $\mathcal{R}$, we have $(\mathcal{E}[N_0,\epsilon];\mathcal{H}[N_0,\epsilon]) = (\mathcal{E}_0;\emptyset)$ for root process ϵ. The state sequence of MRIt is generated as $N_0 \vdash N_1 \vdash \cdots \vdash N_c$. If $\mathcal{E}[N_c,i]$ is empty, the rewrite rules in $\mathcal{H}[N_c,i]$ would be the final hypotheses proved during the whole procedure.

III. Implementation

In this section, we will talk about our implementation of inductive theorem prover called *lz-itp* based on [4] [7] [8] [9] [10] [11]. We use the lazy evaluation schemas of object-oriented and functional programming supported language $Scala$ to build and reuse the classes to organize the term structures, substitutions, nodes, inference rules, etc. Meanwhile, we also follows the discipline of functional programming in coding so that it could be safer and easier to execute the program in a physically parallel computational environment.

The $node$ is a basic unit of MRIt. It is implemented as a class which contains an equation object as a datum and three sets as labels. We also created a class called $nodes$ for the set N of nodes for which several infernoce rules of MRIt are defined. The $index$ of MRIt which corresponding to the process running RI procedure is implemented as a class containing a sequence of natural numbers holding a lazy hash code to gain efficiency during the numerous index comparisons.

The rule FORK is the key to cover all paralleled processes running with different states. Note that due to the nondeterministic choices of contexts (e.g., which direction to orient, which subterm to be expanded or which rewrite strategy should be applied), we cannot decide the number of processes and strategies statically. Therefore, we do not fix the number of processes in the new procedure, and allow it to dynamically change. When a process encounters n nundeterministic choices, we have it forked into n different

ISBN: 978-988-19253-6-7
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

processes, with each process associated with one of the choices. The *fork* function ψ maps each process index to a natural number which represents the number of processes to be created from the given process by the fork operation. The fork function over a given set P of processes, denoted by ψ_P is defined as follows:

$$\psi_P(p) = \begin{cases} \{p1, p2, \ldots, p\psi(p)\}, & \text{if } p \in P, \\ \{p\}, & \text{otherwise.} \end{cases}$$

This function will be used to fork all processes in P,while remaining other processes untouched.

For example, if a process with the index

$$p = [a_1 a_2 \ldots a_k]$$

have n possible choices of contexts, we have it forked into n processes as:

$$[a_1 a_2 \ldots a_k 1], [a_1 a_2 \ldots a_k 2], \ldots, [a_1 a_2 \ldots a_k n].$$

Based on the label representation, we can simulate the fork operation by replacing the label p in the labels of all nodes with the set of n identifiers $p_1, \ldots, p_n$. In practice, we embed this fork operation into other operations if necessary.

The operation $expand(N, N', n)$ is the core of the whole procedure. Let $n = \langle s : t, H_1, H_2, E \cup E' \rangle$, the operation applies the EXPAND rule of RI in all processes of E' that can orient the equation $s = t$ from left to right. The set E' is moved from the third label to the first in n since in each process in E' the conjecture $s = t$ is removed and the new hypothesis $s \rightarrow t$ is added after the expansion. Moreover, for each new conjecture $s' = t'$ in $\text{Expd}_u(s, t)$, a new node $\langle s' : t', \emptyset, \emptyset, E' \rangle$ is created in order to store the conjecture in the processes of E'. Note that (1) the direction of orientation and (2) the choice of the basic subterm to be expanded are two kinds of nondeterministic choices. Therefore there are two possible fork operations, where one is that to fork the original index $p \in N \cup N'$ into $p1$ and $p2$ by different choice of orienting directions (i.e., left to right or right to left), another is to fork the index p' into $p'1, \ldots, p'k$, if term s have k basic subterms to expand. In our implementation, we follow the discipline of functional programming by never mutating the nodes. We just update them from outside. This means the method needs to return the intermediate results as fresh sets of nodes. The result is structured as a tuple $\langle \mathcal{F}_1, \mathcal{F}_2, \mathcal{M}, \mathcal{C} \rangle$ where:

- $\mathcal{F}_1$: the forked nodes from N (i.e., the labels of original nodes in N are forked into new labels depends on the nondeterministic choices)
- $\mathcal{F}_2$: the forked nodes from N' (i.e., the labels of original nodes in N' are forked into new labels depends on the nondeterministic choices)
- $\mathcal{M}$: the nodes "modified" during $expand(N, N', n)$ operation (i.e., a set only contains one node n' which holds the same datum $s : t$ but modified labels)
- $\mathcal{C}$: the nodes newly created by $expand(N, N', n)$ operation (i.e., the nodes containing new conjectures)

The SIMPLIFY_R rule applies the SIMPLIFY rule of RI using a rewrite in the equational axiom $\mathcal{R}$, which is common to all processes. E is the set of all processes that have $s = t$ as a conjecture. Since this equation is transformed to an equation $s' = t$, the set E is removed from the original node, and a new node $\langle s' : t, \emptyset, \emptyset, E \rangle$ is created. The SIMPLIFY_H rule is almost the same as SIMPLIFY_R. The difference is that SIMPLIFY_R applies a rule of $\mathcal{R}$, while SIMPLIFY_H applies an inductive hypothesis of $\mathcal{H}$, which may exist only in some distinguished processes. This makes the third labels of the original node and the new node $E \backslash H$ and $E \cap H$, respectively.

The operation $simplify(N, N', n)$ applies the rule SIMPLIFY_R and SIMPLIFY_H to n as much as possible. Note that in the original MRIt, the two rules are defined separately. However, in our implementation we combine the two rules into one operation because we have to fork the other nodes N and N' at the same time. The rewrite strategy often plays an important role in simplification [7] [13] [15], therefore we fork the original index $p \in N \cup N'$ into $p1, \ldots, pk$, if there are k normal forms generated by different strategies (e.g., outermost and innermost strategy). Like the result tuple of EXPAND operation, the result of SIMPLIFY operation is also structured as a tuple $\langle \mathcal{F}_1, \mathcal{F}_2, \mathcal{M}, \mathcal{N} \rangle$ where $\mathcal{N}$ stands for the nodes newly created by $simplify(N, N', n)$ (i.e., the nodes containing rewritten term s' with modified labels).

The operation $N.delete()$ removes from N all nodes that contain a trivial equation, and returns the remaining nodes as N'. This operation would be applied to the nodes created by rules SIMPLIFY_R, SIMPLIFY_H and EXPAND.

The operation $N.gc()$ implements the rule GC of MRIt, removes the nodes with three empty labels. It can effectively reduce the size of the current node database by removing nodes with three empty labels, because no processes contain the corresponding rule or equation.

The operation $N.subsume()$ combines two nodes into a single one when they contain the variant data (which are the same as each other up to renaming of variables). The duplicate indices in the third labels are removed to preserve the label conditions. The operation $N.subsume()$ is invoked by the operation $union(N, N')$ which is designed for combining nodes N and N'. We exploited the same *lazy* technique as [9] [10] to gain efficiency by creating a hash map $[\mathcal{I}_s, \mathcal{N}]$, where $\mathcal{N}$ is a *list* of nodes and $\mathcal{I}_s$ is a lazy value defined in the node class as the *size* of the node, so that we need only check the nodes with the same size as the original nodes. This check can be done effciently by using the hash map with the node size as its key. In other words, for every $n \in N$, n uses its size $\mathcal{I}_n$ as the key to $[\mathcal{I}_s, N]$, then the set $\mathcal{N}_n$ containing all the nodes with same size $\mathcal{I}_n$ is looked up for the nodes with variant data.

The operation $N.subsume\_P()$ stops redundant processes, which have the same state as other existing processes. The function $sub(N, L)$ is defined as $sub(N, L) = \{\langle s : t, H_1 \backslash L, H_2 \backslash L, E \backslash L \rangle \mid \langle s : t, H_1, H_2, E \rangle \in N\}$, it simply removes all indices in L from every node in N.

The operation $lemmaExplore(N, n)$ is a new introduced operation of MRIt. In term rewriting induction, it is well-known that it is effective for proving some problems by supplying appropriate lemmas. We exploit the lemma exploration by divergence analyzation [12], which is a powerful lemma generation method. We consider with the pseudocode of our implementation in *Algorithm 1*. The procedure is based on the open/closed (set-of-support/have-been-given) lists algorithm, which is well-known in the literature of search and automated reasoning for artificial intelligence.

ISBN: 978-988-19253-6-7
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

When a node $n = \langle s : t, H_1, H_2, E\rangle$[1] is simplified (both s and t are the normal forms in the corresponding processes) and expanded (line 24). We put it into N_c as the hypothesis and try to analyze all processes P holds by n (line 28). It is not efficient to directly analyze all process that n covers. Because although after operation $N.subsum\_P()$, the state $(\mathcal{E}[N_c, i], \mathcal{H}[N_c, i])$ becomes unique, there may still exist duplicate projections where $\mathcal{H}[N_c, i] = \mathcal{H}[N_c, j]$. We created a hash map $[\mathcal{S}_i, \mathcal{L}_i]$ in order to deal with the lemma generations in every process $i \in P$, where $\mathcal{S}_i$ indicates the hash code of $\mathcal{H}[N_c, i]$ (we created an class *projection* for projections, where contains the lazy hash code) and $\mathcal{L}_i$ denotes the result of possible lemmas. Since the key is unique in a hash map, we filter the duplicate keys easily by creating the map. The lemma generation function scans every set of hypotheses in different processes corresponding to the keys as the values of hash map $[\mathcal{S}_i, \mathcal{L}_i]$. Finally, the new nodes as possible lemmas in corresponding processes of n are constructed, then they are put into N_o (line 28).

The procedure $success(N_o, N_c)$ checks if this induction procedure has succeeded. The process succeeds if there exists an index $i \in I$ such that i is not contained in any labels of N_o and any E labels of N_c nodes. Then $\mathcal{E}[N_o \cup N_c, i] = \emptyset$, and $\mathcal{R}[N_c, i]$ is a set of rewrite rules as the final hypotheses which have been proved. The proof details will also be captured by the program as an output.

Note that in line 22 to 26 of *Algorithm 1*, we apply SUBSUME_P rule of MRIt to N_o, N_c and n by the same context. Which means we should implicitly subsume the same duplicate indices L (depends on their states) with N_o, N_c and n (i.e., $N_o = sub(N_o, L), N_c = sub(N_c, L), n = sub(\{n\}, L).head$). For the same reason, we build $simplify(N_o, N_c, n)$ and $expand(N_o, N_c, n)$ to take three parameters in order to fork nodes form N_o and N_c at the same time.

The $N.choose()$ always choose the minimal node in terms of its size which makes the computation efficient. And there is another heuristic idea in our implementation that different from the original MRIt. We try to simplify the conjectures at the very first before we expand them. Because we found that some inductive theorems are often reducible to the given TRS. We are not sure if this will make the proofs itself shorter (because in some cases it does while others not), however in many cases observed in our experiments, this will reduce the choices of nodes as well as the scale of whole node database.

IV. Experiment

In this section, we talk about some experimental results. In the implementation of *lz-itp*, we used a built-in termination checker (developed by ourselves) based on the dependency-pair method [17] [18] [19]. We also used the combination of polynomial interpretation and SAT solving as proposed in [20] in order to find reduction orders for ensuring termination. All experiments were performed on a PC with i5 CPU and 4GB main memory.

[1] $n.mir$ represents the symmetric case of n

Algorithm 1 lz-itp($\mathcal{E}, \mathcal{R}$)

1: $N_o := \{\langle s : t, \emptyset, \emptyset, \{\epsilon\}\rangle \mid s \leftrightarrow t \in \mathcal{E}\}$
2: $N_c := \emptyset$
3: **while** $success(N_o, N_c) = false$ **do**
4: **if** $N_o = \emptyset$ **then**
5: return(fail)
6: **else**
7: $n := N_o.choose()$
8: $\langle \mathcal{F}_1, \mathcal{F}_2, \mathcal{M}, \mathcal{N}\rangle := simplify(N_o, N_c, n)$
9: $N_o := union(N_o - \{n\}, \mathcal{N}.delete())$
10: $N_o := \mathcal{F}_1$
11: $N_c := \mathcal{F}_2$
12: $n := \mathcal{M}.head$
13: **if** $n \neq \langle \ldots, \emptyset, \emptyset, \emptyset\rangle$ **then**
14: **if** $n \neq \langle \ldots, \emptyset, \emptyset, \ldots\rangle$ **then**
15: $\langle \mathcal{F}_1, \mathcal{F}_2, \mathcal{M}, \mathcal{N}\rangle := simplify(N_o, N_c, n.mir)$
16: $N_o := union(N_o, \mathcal{N}.delete())$
17: $N_o := \mathcal{F}_1$
18: $N_c := \mathcal{F}_2$
19: $n := \mathcal{M}.head$
20: $\langle \mathcal{F}_1, \mathcal{F}_2, \mathcal{M}, \mathcal{C}\rangle := expand(N_o, N_c, n)$
21: $N_o := union(N_o, \mathcal{C}.delete())$
22: $N_o := \mathcal{F}_1.subsume\_P()$
23: $N_c := \mathcal{F}_2.subsume\_P()$
24: $n := \mathcal{M}.head$
25: **end if**
26: $n := n.subsume\_P()$
27: $N_c := union(N_c, \{n\}).gc()$
28: $N_o := union(N_o, lemmaExplore(N_c, n))$
29: **end if**
30: **end if**
31: **end while**
32: return $\mathcal{H}[N_c, i]$ where $i = success(N_o, N_c)$

First we consider a propositional logic problem from [21]:

$$\mathcal{R} = \begin{cases} not(T) \to F \\ not(F) \to T \\ and(T, p) \to p \\ and(F, p) \to F \\ or(T, p) \to T \\ or(F, p) \to p \\ implies(p, q) \to or(not(p), q) \end{cases}$$

We prove the theorem

$$implies(and(p, q), or(p, q)) = T$$

by at least two EXPAND operations. It is obvious that the left-hand side of the theorem can be rewritten to

$$or(not(and(p, q)), or(p, q))$$

by the last rule $implies(p, q) \to or(not(p), q)$ first. Then it can be expanded to

$$or(not(and(T, p)), T) = T,$$

$$or(not(and(F, p)), p) = T,$$

where the first conjecture will be rewritten to $or(not(p), T) = T$ which needs the second expansion to finish the proof.

ISBN: 978-988-19253-6-7
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

We can also expand the original target directly into

$$implies(p, or(T, p)) = T,$$
$$implies(F, or(F, p)) = T.$$

After the simplification of the first conjecture we will still get

$$or(not(p), T) = T$$

to be ready for the second expansion. As we can see, although the length of the proof did not change, in our program, the first method checked 7 nodes with one succeeded process over 2 processes and the second checked 11 nodes with 2 succeeded processes over 6 processes.

Some other problems form [6] [21] also showed the similar performance summarized in the TABLE I and TABLE II:

TABLE I
SIMPLIFY FIRST

problem	time (ms)	# of nod.	# of succ.	# of proc.
ex_1	17988	263	9	105
ex_2	6706	305	2	32
ex_3	1108	37	2	9

TABLE II
EXPAND FIRST

problem	time (ms)	# of nod.	# of succ.	# of proc.
ex_1	47866	276	18	223
ex_2	7075	398	2	26
ex_3	1322	57	2	11

where "# of nod." shows the number of processed nodes when the procedure succeed; "# of succ." shows the number of succeeded processes on average during the computation; "# of proc." shows the number of all processes when a process has succeeded. We can see that in these problems, the "simplify first" strategy could reduce the number of processed nodes so that the computation time of the whole procedure was also reduced.

In our implementation, the SIMPLIFY operation tries two rewrite strategies: the leftmost innermost strategy and the leftmost outermost strategy. Since MRIt also works on the set of nodes, we exploited the lazy evaluation scheme for the nodes manipulation proposed in [9] [10] to gain more efficiency. Moreover, we implemented the lemma exploration function with divergence analyzation [12] to deal with the lemma-required problems. The problems selected from [6] which need appropriate lemmas were examined as shown in TABLE III.

TABLE III
EXPAND FIRST

problem	lem_1	lem_2	lem_3	lem_4	lem_5	lem_6
lz-itp	602	932	12379	17738	1050	1023
mrit+	615	969	12801	18090	1075	1049
mrit	-	-	12952	-	-	-

Note that the *mrit+* in TABLE III stands for an implementation of MRIt with the lemma exploration, while the *mrit* stands for the original implementation of MRIt. The *mrit* failed in most of the cases with a time limit in 60000ms. We can see *lz-itp* which used the lazy evaluation schemas was more efficient than *mrit+*.

V. Conclusion

We have presented a new implementation of the multi-context algebraic inductive theorem prover *lz-itp* based on the multi-context rewriting induction. We applied lazy evaluation schemas with several heuristic ideas in our implementation. The experiments show that *lz-itp* was more efficient than MRIt in all the problems examined. To implement and examine with more powerful lemma exploring methods is a possible work in future. To study extensions for handling non-orientable equations is also an interesting future work.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Number 25330074.

References

[1] L. Bachmair, *Canonical Equational Proofs*. Birkhäuser, 1991.
[2] N. Dershowitz and J.-P. Jouannaud, "Rewrite Systems," in J. van Leeuwen (ed.), *Handbook of Theoretical Computer Science*, Vol.B, North-Holland, 1990, pp.243-320.
[3] D. A. Plaisted, "Equational reasoning and term rewriting systems," in D. M. Gabbay et al. (eds.), *Handbook of Logic in Artificial Intelligence and Logic Programming*, Vol. 1, Oxford Univ. Press, 1993, pp.274-367.
[4] U. Reddy, "Term rewriting induction," *10th Int. Conf. on Automated Deduction, vol.814 of Lecture Notes in Computer Science*, pp.162—177, 1990.
[5] T. Aoto, "Rewriting induction using termination checker," *JSSST 24th Annual Conference*, 3C-3, 2007 (in Japanese).
[6] T. Aoto, "Dealing with Non-orientable Equations in Rewriting Induction," *Proc. 17th International Conference on Rewriting Techniques and Applications*, vol. 4098 of Lecture Notes in Computer Science, pp.242—256, 2006.
[7] H. Sato, M. Kurihara, "Multi-Context Rewriting Induction with Termination Checkers," *IEICE Transactions on Information and Systems*, Vol. E93.D, No.5, 2010, pp.942-952.
[8] M. Kurihara and H. Kondo, "Completion for multiple reduction orderings," *Journal of Automated Reasoning*, Vol.23, No.1, 1999, pp.25-42.
[9] CC. Ji, H. Sato, M. Kurihara, "An Efficient Implementation of Multi-Context Algebraic Reasoning System with Lazy Evaluation," *Lecture Notes in Engineering and Computer Science: Proceedings of The International MultiConference of Engineers and Computer Scientists 2015*, IMECS 2015, 18-20 March, 2015, Hong Kong, pp.201-205.
[10] CC. Ji, H. Sato, M. Kurihara, "Lazy Evaluation Schemes for Efficient Implementation of Multi-Context Algebraic Completion System," *IAENG International Journal of Computer Science*, vol. 42, no. 3, pp.282-287, 2015.
[11] H. Sato, "Effectiveness of Context-Search in Rewriting Induction," *IPSJ Special Interest Group on Programming 2011*, 14-15 Jun. 2011 (in Japanese).
[12] T. Walsh, "A divergence critic for inductive proof", *Journal of Artificial Intelligence Research*, vol. 4, pp. 209—235, 1996.
[13] K. Hirotaka, T. Yoshihito, "Inductionless Induction and Rewriting Induction," *Computer Software,* vol.17, No.6, pp.1-12, 2000 (in Japanese).
[14] G. Huet and D. C. Oppen, "Equations and rewrite rules: A survey," in R. Book (ed.), *Formal Language Theory: Perspectives and Open Problems*, Academic Press, 1980, pp.349-405.
[15] G. Huet and J.-M. Hullot, "Proofs by induction in equational theories with constructor," *Journal of Computer and System Sciences*, vol.25, no.2, pp.239—266, 1982.
[16] Terese, *Term rewriting systems.* Cambridge University Press. 2003.
[17] T. Arts and J. Giesl, "Termination of term rewriting using dependency pairs," *Theoretical Computer Science*, vol.236, pp.133—178, 2000.
[18] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke, "Mechanizing and improving dependency pairs," *Journal of Automated Reasoning*, vol.37, no.3, pp.155—203, 2006.
[19] N. Hirokawa and A. Middeldorp, "Tyrolean termination tool: techniques and features," *Information and Computation*, vol.205, no.4, pp.474—511, 2007.
[20] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl, "SAT solving for termination analysis with polynomial interpretations," *Proc. 10th International Conference on Theory and Applications of Satisfiability Testing (SAT 2007)*, vol. 4501 of Lecture Notes in Computer Science, pp.340—354, 2007.
[21] R. S. Boyer and J. S. Moore, "A Computational Logic," *Academic Press*, 1979.