Lazy Evaluation Schemes for Efficient Implementation of Multi-Context Algebraic Completion System

ChengCheng Ji, Haruhiko Sato and Masahito Kurihara

***Abstract*—Lazy evaluation is a computational scheme which delays the evaluation of an expression until its value is needed, trying to improve the performance particularly when dealing with large data structure. In this paper, we apply this mechanism to a multi-context algebraic reasoning system which, on a large data structure called the nodes, efficiently simulates parallel processes each executing an algebraic reasoning procedure under a particular context (or a premise). In particular, the multi-completion system MKB simulates the parallel Knuth-Bendix completion procedures, which, given a set of equations and a set of reduction orderings, try to generate a complete (i.e., terminating and confluent) term rewriting system equivalent to the input equations. Exploiting the lazy evaluation, we present an efficient implementation of MKB, called *lz-mkb*, and implement it in a functional, object-oriented programming language Scala which features the lazy evaluation mechanism. The experiments with standard sample problems show that *lz-mkb* is more efficient than the original MKB implementation of Kurihara and Kondo.**

***Index Terms*—Term rewriting system, Completion, Multi-Completion, Kunth-Bendix completion, Lazy evaluation.**

I. Introduction

MULTI-CONTEXT algebraic reasoning systems efficiently simulate parallel processes each executing an algebraic reasoning procedure under a particular context (or a premise). Those systems are used to reason about algebraic computational systems such as term rewriting systems (TRSs), which are a concise and rigorous representation of computational systems in terms of rewrite rules. In fact, TRSs are studied and used in various areas of computer science, including automated theorem proving, analysis and implementation of abstract data types, and decidability of word problems. A TRS is said to be complete if it satisfies the properties called termination and confluence.

The well-known procedure for the completion of TRS was invented by Knuth and Bendix [5] in 1970 and affected a lot of researchers since then. Given a set of equations and a reduction ordering on a set of terms, the procedure (called KB in this paper) uses the ordering to orient equations (either from left to right or from right to left to transform them into rewrite rules) and tries to generate a complete TRS equationally equivalent to the input set of equations. The resultant TRS can be used to decide the equational consequences (word problems) of the input equations.

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.

Actually, however, the KB leads to three possible results: success, failure, or divergence. In the success case, the procedure stops and outputs a complete TRS. In the failure case, the procedure stops but only returns a failure message with an unorientable equation. In the divergence case, the procedure falls into an infinite loop, trying to generate an infinite set of rewrite rules. The result of KB seriously depends on the given reduction ordering. With a good ordering, it would lead to a success, but otherwise, it would cause the failure or the divergence. In the latter case, we could try to avoid them by changing the ordering to appropriate one, but the problem is that it is very difficult for ordinary software designers and AI researchers to design or choose an appropriate ordering.

Therefore, automatic search for appropriate orderings is desired. But according to the possibility of divergence, we cannot try candidate orderings one by one. Also, it is not efficient to simply create processes for each different ordering and run them in parallel on a machine, because the number of candidate orderings normally exceeds ten thousands even for a small problem.

In 1999, this problem was partially solved by a completion procedure called MKB [6]. MKB is a single procedure that efficiently simulates execution of multiple processes each running KB with a different reduction ordering. The key idea of MKB lies in a data structure called node. The node contains a pair $s : t$ of terms and three sets of indices to orderings to show whether or not each process contains rules $s \to t$, $t \to s$, or an equation $s = t$. The well-designed inference rules of MKB allows an efficient simulation of multiple inferences in several processes all in a single operation.

In this paper, we present an efficient implementation of MKB, called *lz-mkb*, by exploiting the lazy evaluation schemes. The lazy evaluation, sometimes called the call-by-need, is a computational scheme which delays the evaluation of an expression until its value is needed and which also avoids repeated evaluations by the 'memoization' mechanism to share the common computational results. Thus the lazy evaluation can lead to the improvement of performance by avoiding needless calculations particularly when dealing with a large data structure with compound objects. Noting that MKB works on a large data structure of nodes, we introduced the lazy evaluation scheme into MKB to develop *lz-mkb*. The actual implementation of *lz-mkb* is written in Scala, a rising programming language supporting both functional and object-oriented programming, featuring the lazy evaluation.

This paper is organized as follows. In Section II, we will provide a brief review on TRSs and completion procedures KB and MKB. In Section III, we will discuss the implementation of *lz-mkb*. The result of the experiments will be shown and discussed in Section IV. In Section V, we will conclude with possible future work. This paper is an extension of our preliminary work [4] with additional experiments and analyses.

II. Preliminaries

A. Term Rewriting Systems

Let us briefly review the basic notions for term rewriting systems(TRS) [1] [2] [3] [8] [12]. 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 $s \equiv t$ 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.

When we use TRS to solve specified problems, some properties such as $termination$ and $confluence$ are expected to hold most of the time. To talk about those properties, we need more definitions as follows.

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 $C[r\sigma] \equiv t$. 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 $l \to r \in R$.

After termination we talk about confluence, which is also an important property often expected.

Definition 2.7: Two terms s, t in TRS R are *joinable* (notation $s \downarrow t$), if there exists a term v such that $s \to_R^* v$ and $t \to_R^* v$, where $\to_R^*$ is the reflexive transitive closure of $\to_R$.

Theorem 2.8: A TRS R is *confluent* iff for all terms $s, t, u \in T(\Sigma, V)$, $u \to_R^* s$ and $u \to_R^* t$ implies $s \downarrow t$.

Definition 2.9: The *composition* $\sigma\tau$ of two substitutions σ and τ is defined as $s(\sigma\tau) = (s\sigma)\tau$. A substitution σ is $more\ general$ than a substitution σ' if there exists a substitution δ such that $\sigma' = \sigma\delta$. For two terms s and t, if there is a substitution σ such that $s\sigma \equiv t\sigma$, σ is a unifier of s and t. We denote the most general unifier of s and t by $mgu(s, t)$.

With *Definition 2.9* we can define $critical\ pairs$ as follows:

Definition 2.10: Consider two rewrite rules $l_1 \to r_1$ and $l_2 \to r_2$ in a TRS R with no common variables. (If they have common variables, we can rename them properly.) If a term s is a subterm of l_1 denoted by $l_1[s]$, and if there exists an $mgu(s, l_2) = \sigma$, then the pair $\langle l_1\sigma[r_2\sigma], r_1\sigma \rangle$ of terms is called a *critical pair* of $l_1 \to r_1$ and $l_2 \to r_2$.

For example, let f be a function symbol, $\{a, b, c\}$ be variables, and consider two rewrite rules $f(a) \to b$ and $a \to c$. By setting $s = a$ (the argument of $f(a)$) and $l_2 = a$ (the left-hand side of the second rule) , we have the empty mgu (or the identical mapping, meaning that no variables need to be replaced). Since $l_1[r_2] = f(c)$ and $r_1 = b$, we obtain $\langle f(c), b \rangle$ as a critical pair. In TRS, confluence can be decided with $critical\ pairs$.

Theorem 2.11: A terminating TRS is confluent iff all critical pairs (p, q) satisfy $p \downarrow q$.

If a TRS R satisfies termination and confluence, we say R is *complete* (or *convergent*) or R has the $completion$ property.

B. Completion procedure

To complete a TRS, we need some procedures. Here we will talk about the standard completion procedure KB and multi-completion procedure MKB [5] [6].

Given a set of equations $\mathcal{E}_0$ and a reduction ordering $\succ$, the standard completion procedure ***KB*** tries to generate a convergent set $\mathcal{R}_c$ of rewrite rules that is contained in $\succ$ and that induces the same equational theory as $\mathcal{E}_0$. The KB procedure implements the following six inference rules.

DELETE: $(\mathcal{E} \cup \{s \leftrightarrow s\}; \mathcal{R}) \vdash (\mathcal{E}; \mathcal{R})$

COMPOSE: $(\mathcal{E};\mathcal{R}\cup\{s\to t\})\vdash(\mathcal{E};\mathcal{R}\cup\{s\to u\})$
if $t\to_{\mathcal{R}} u$

SIMPLIFY: $(\mathcal{E}\cup\{s\leftrightarrow t\};\mathcal{R})\vdash(\mathcal{E}\cup\{s\leftrightarrow u\};\mathcal{R})$
if $t\to_{\mathcal{R}} u$

ORIENT: $(\mathcal{E}\cup\{s\leftrightarrow t\};\mathcal{R})\vdash(\mathcal{E};\mathcal{R}\cup\{s\to t\})$
if $s\succ t$

COLLAPSE: $(\mathcal{E};\mathcal{R}\cup\{t\to s\})\vdash(\mathcal{E}\cup\{u\leftrightarrow s\};\mathcal{R})$
if $l\to r\in\mathcal{R},\ t\to_{\{l\to r\}} u$, and $t\rhd l$

DEDUCE: $(\mathcal{E};\mathcal{R})\vdash(\mathcal{E}\cup\{s\leftrightarrow t\};\mathcal{R})$
if $u\to_{\mathcal{R}} s$ and $u\to_{\mathcal{R}} t$

The new symbol $\rhd$ here denotes the *encompassment ordering* defined as follows.

Definition 2.12: An *encompassment order* $\rhd$ on a set of terms is defined by $s\rhd t$ iff some subterm of s is an instance of t and $s\not\doteq t$.

For example, if $\{f,g\}$ are function symbols and $\{x,y,z\}$ variables, then $f(x,g(x))\rhd f(y,g(z))$ but $f(x,g(y))\not\rhd f(z,g(z))$. KB starts from the initial state $(\mathcal{E}_0,\mathcal{R}_0)$ where $\mathcal{R}_0=\emptyset$. The procedure changes the states in a possibly infinite completion sequence $(\mathcal{E}_0;\mathcal{R}_0)\vdash(\mathcal{E}_1;\mathcal{R}_1)\vdash\cdots$ by its inference rules. The result of the completion sequence is the sets $\mathcal{E}_c$ and $\mathcal{R}_c$. When $\mathcal{E}_c=\emptyset$, $\mathcal{R}_c$ will be a confluent and terminating TRS satisfying $\leftrightarrow^*_{\mathcal{R}_c}=\leftrightarrow^*_{\mathcal{E}_0}$, which means KB procedure has succeeded. And the sequence has failed if $\mathcal{E}_c\neq\emptyset$.

A completion procedure for multiple reduction orderings called *MKB* developed in [6] accepts a finite set of reduction orderings $O=\{\succ_1,\ldots,\succ_n\}$ and a set of equations $\mathcal{E}_0$ as input. The proper output is a set of a convergent rewrite rules $\mathcal{R}_c$. To achieve the multi-completion, MKB effectively simulates KB procedures in n parallel processes $\{P_1,\ldots,P_n\}$ corresponding to O. Let $I=\{1,\ldots,n\}$ be the index set and $i\in I$ be an index. In this setting, P_i executes KB for the reduction order $\succ_i$ and the common input $\mathcal{E}_0$. The inference rules of MKB which simulate the related KB inferences all in a single operation is based on a special data structure called the *node* defined below.

Definition 2.13: A *node* is a tuple $\langle s:t,R_0,R_1,E\rangle$, where $s:t$ is an ordered pair of terms s and t called *datum*, and R_0,R_1,E are subsets of I called *labels* such that:

- R_0,R_1 and E are mutually disjoint. (i.e., $R_0\cap R_1=R_0\cap E=R_1\cap E=\emptyset$)
- $i\in R_0$ implies $s\succ_i t$, and $i\in R_1$ implies $t\succ_i s$

Intuitively, the set $R_0(R_1)$ represents the indices of processes executing KB in which the set of rewrite rules $\mathcal{R}$ currently contains $s\to t$ ($t\to s$), and E represents those of processes in which $\mathcal{E}$ contains an equation $s\leftrightarrow t$ (or $t\leftrightarrow s$). The node $\langle s:t,R_0,R_1,E\rangle$ is considered to be identical with the node $\langle t:s,R_1,R_0,E\rangle$, hence the inference rules of MKB working on a set N of nodes defined below implicitly specify the symmetric cases.

DELETE: $N\cup\{\langle s:s,\emptyset,\emptyset,E\rangle\}\vdash N$
if $E\neq\emptyset$

ORIENT: $N\cup\{\langle s:t,R_0,R_1,E\cup E'\rangle\}\vdash$
$N\cup\{\langle s:t,R_0\cup E',R_1,E\rangle\}$
if $E'\neq\emptyset, E\cap E'=\emptyset$,
and $s\succ_i t$ for all $i\in E'$

REWRITE_1: $N\cup\{\langle s:t,R_0,R_1,E\rangle\}\vdash$
$N\cup\left\{\begin{array}{l}\langle s:t,R_0\backslash R,R_1,E\backslash R\rangle\\ \langle s:u,R_0\cap R,\emptyset,E\cap R\rangle\end{array}\right\}$
if $\langle l:r,R,\ldots,\ldots\rangle\in N,\ t\to_{\{l\to r\}} u$,
$t\doteq l$, and $(R_0\cup E)\cap R\neq\emptyset$

REWRITE_2: $N\cup\{\langle s:t,R_0,R_1,E\rangle\}\vdash N\cup$
$\left\{\begin{array}{l}\langle s:t,R_0\backslash R,R_1\backslash R,E\backslash R\rangle\\ \langle s:u,R_0\cap R,\emptyset,(R_1\cup E)\cap R\rangle\end{array}\right\}$
if $\langle l:r,R,\ldots,\ldots\rangle\in N,\ t\to_{\{l\to r\}} u$,
$t\rhd l$, and $(R_0\cup R_1\cup E)\cap R\neq\emptyset$

DEDUCE: $N\vdash N\cup\{\langle s:t,\emptyset,\emptyset,R\cap R'\rangle\}$
if $\langle l:r,R,\ldots,\ldots\rangle\in N$,
$\langle l':r',R',\ldots,\ldots\rangle\in N, R\cap R'\neq\emptyset$,
and $s\leftarrow_{\{l\to r\}} u\to_{\{l'\to r'\}} t$

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

SUBSUME: $N\cup\left\{\begin{array}{l}\langle s:t,R_0,R_1,E\rangle\\ \langle s':t',R'_0,R'_1,E'\rangle\end{array}\right\}\vdash$
$N\cup\{\langle s:t,R_0\cup R'_0,R_1\cup R'_1,E''\rangle\}$
if $s:t$ and $s':t'$ are variants and
$E''=(E\backslash(R'_0\cup R'_1))\cup(E'\backslash(R_0\cup R_1))$

Given the current set N of nodes, $(E[N,i];R[N,i])$ defined in the following represents the current set of equations and rewrite rules in a process P_i.

Definition 2.14: Let $n=\langle s:t,R_0,R_1,E\rangle$ 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 equations defined by

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

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

$$\mathcal{R}[n,i]=\begin{cases}\{s\to t\}, & \text{if } i\in R_0,\\ \{t\to s\}, & \text{if } i\in R_1,\\ \emptyset, & \text{otherwise.}\end{cases}$$

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

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

MKB starts with the initial set N_0 of nodes:

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

which means, given the initial set of equations $\mathcal{E}_0$, we have $(\mathcal{E}[N_0,i];\mathcal{R}[N_0,i])=(\mathcal{E}_0;\emptyset)$ for all $i\in I$. The state sequence of MKB is generated as $N_0\vdash N_1\vdash\cdots\vdash N_c$. If $\mathcal{E}[N_c,i]$ is empty and all critical pairs of $\mathcal{R}[N_c,i]$ have been created, MKB returns $\mathcal{R}[N_c,i]$ as the result, which is a convergent TRS obtained by a successful KB sequence in the process P_i.

III. Implementation

In this section we will discuss the details about the implementation. We implemented an algebraic reasoning system called *lz-mkb* based on MKB in [6] by using lazy evaluation mechanism of the programming language *Scala*. Scala is a programming language which supports *functional programming* and *object-oriented programming*. The program was designed in an object-oriented way so that we could build and reuse the classes to organize the term structures, subsitutions, nodes, inference rules, etc. At the same time, we also followed the discipline of functional programming (e.g., "uniform return type" principle [7]) in coding so that it could be safer and easier to execute the program in a physically parallel computational environment.

The node, a basic unit of MKB, is implemented as a class which contains an equation object as a datum and three $bitsets$ as labels. We chose bitset[1]to gain efficiency because there were numerous set operations during the computation. We also created a class called *nodes* for the set N of nodes for which several inference rules of MKB are defined. We will discuss the implemented operations below in comparison with the original inference rules of MKB one by one.

The operation *N.delete()* simply removes from N all nodes that contain a trivial equation, and returns the remaining nodes as N'. This operation is only applied to the nodes created by rules REWRITE and DEDUCE.

The operation *n.orient()* orients the equation from left to right or right to left by changing their labels from E to R_0 or E to R_1 according to the reduction order in each process. Notice that the application of the reduction order to an equation should be done twice (i.e., one with $s : t$ and one with $t : s$) in theory, but in practice we implemented it so that it was executed only once, noting that at most one of them should be true. The indices still remaining after this operation in E correspond to the reduction orders that failed to orient the equation.

The operation *rewrite(N, N')* is not included in the class of nodes but it takes nodes as arguments. In the original idea of MKB, REWRITE_1 and REWRITE_2 simulate the COMPOSE, SIMPLIFY and COLLAPSE (if appropriate conditions are satisfied) in one single operation. More exactly, REWRITE_1 and REWRITE_2 are repeatedly applied to $N \cup N'$, rewriting the data of N by the rules of N' until no more rewriting is possible. It returns the set of nodes created in this process and the mutation operations are applied to N so that N is updated as

$$N := N - \{\text{original nodes}\} \cup \{\text{updated nodes}\}.$$

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 $\langle \mathcal{D}, \mathcal{N}, \mathcal{M} \rangle$ where:

- $\mathcal{D}$: the nodes rewritten by rewrite(N, N')(i.e., the original ones with the original datum $s : t$)
- $\mathcal{N}$: the nodes "created" by rewrite(N, N')(i.e., the new nodes with the original datum $s : t$ and updated labels)
- $\mathcal{M}$: the nodes "modified" during rewrite(N, N')(i.e., the new nodes with a new datum $s : u$ and updated labels)

Normally, after the *rewrite(N, N')* operation, N should be updated as $N := N + \mathcal{M} - \mathcal{D}$. If N only has one node in it (i.e., $N = \{n\}$), the modified n would be returned by $\mathcal{M}.head$.

Notice that to the symmetric cases of nodes, we just use the *mirrors* which refer to the symmetric nodes of the original N and N' as input. In other words, in every one-step rewrite, we need to do this operation four times with different combinations from $\{(N, N'), (N.mir, N'.mir), (N.mir, N'), (N, N'.mir)\}$ one by one. Surely (N, N') is updated after every single rewrite_1 or rewrite_2. In this way, we obtain a tuple $\langle \mathcal{D}_\infty, \mathcal{N}_\infty, \mathcal{M}_\infty \rangle$ of three nodes in which every calculated node is included and no more rewrite can be applied. Finally, the tuple $\langle \mathcal{D}_\infty, \mathcal{N}_\infty - \mathcal{D}_\infty, \mathcal{M}_\infty - \mathcal{D}_\infty \rangle$ is returned as the result of the operation rewrite(N, N').

The operation $N.deduce(n)$ generates all the possible critical pairs between n and $\{n\} \cup N$. We consider all combinations of pair of nodes. For example, consider two nodes $n = \langle a : b, R_0, R_1, \dots \rangle$ and $n' = \langle c : d, R'_0, R'_1, \dots \rangle$. The operation $\{n\}.deduce(n')$ considers the critical pairs from $\{a \leftrightarrow b, c \leftrightarrow d\}$, which means the modification of labels should be considered for each of $\{R_0 \cap R'_0, R_1 \cap R'_0, R_0 \cap R'_0, R'_1 \cap R'_0\}$.

The operation $N.garbagecollect()$ has no related inference rules in KB. In MKB, 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. We exploited a programming technique called *lazy evaluation* to gain efficiency in the implementation. To discuss the details, we consider with the pseudocode of implementation presented as *Algorithm 1*, based on the presentation in [4]. The operation $N.subsume()$ is invoked by the operation $union(N, N')$ which is designed for combining nodes N and N'. We observe that in every iteration of the *while loop*, the $union(N, N')$ operation is called at least once (i.e., for every chosen n, *subsume()* would be called at *line 9* once; And for those that satisfied the proper conditions of *line11* and *line 13*, two more operations are required). This means *subsume()* would be invoked frequently during the whole procedure. It would make the program slower to simply check all of the nodes in N, when N was updated after rewrite operations. To gain efficiency, we created a $lazy$ 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 (i.e., for a node $n = \langle s : t, r_0, r_1, e \rangle$, n.size $= s$.size $+ t$.size), so that we need only check the nodes with the same size as the original nodes. This check can be done efficiently 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 $[\mathcal{I}_s, \mathcal{N}]$, 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. In our Scala program, the hash map $[\mathcal{I}_s, N]$ is declared to be *lazy*, because it is calculated only once and then be stored as a constant object ready to be returned for repeated calculation requests afterwards.

Notice that the procedure $success(N_o, N_c)$ checks if this completion process 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 convergent set of rewrite rules contained in $\succ_i$. We also created lazy values in nodes

[1]a data structure defined in Scala's library

Algorithm 1 lz-mkb(E, O)

$N_o := \{\langle s : t, \emptyset, \emptyset, I\rangle \mid s \leftrightarrow t \in E\}$ where $I = \{1, \ldots, |O|\}$
$N_c := \emptyset$
while $success(N_o, N_c) = false$ **do**
  if $N_o = \emptyset$ **then**
    return(fail)
  else
    $n := N_o.choose()$
    $\langle \mathcal{D}, \mathcal{N}, \mathcal{M}\rangle := rewrite(\{n\}, N_c)$
    $N_o := union(N_o - \{n\}, \mathcal{N}.delete())$
    $n := \mathcal{M}.head$
    if $n \neq \langle \ldots, \emptyset, \emptyset, \emptyset\rangle$ **then**
      $n := n.orient()$
      if $n \neq \langle \ldots, \emptyset, \emptyset, \ldots\rangle$ **then**
        $\langle \mathcal{D}, \mathcal{N}, \mathcal{M}\rangle := rewrite(N_c, \{n\})$
        $N_o := union(N_o, \mathcal{N}.delete())$
        $N_c := N_c + \mathcal{M} - \mathcal{D}$
        $N_c := N_c.garbagecollect()$
        $N_o := union(N_o, deduce(n, N_c).delete())$
      end if
      $N_c := union(N_c, \{n\})$
    end if
  end if
end while
return $\mathcal{R}[N_c, i]$ where $i = success(N_o, N_c)$

to hold the occurrences of the index i in the labels, so that we do not need to calculate it in the unchanged N_c every time. This also makes the computation efficient as $N.choose()$ operation will always choose the minimal node in terms of its size.

IV. Experiment

In this section, we will show how the program performed with the lazy evaluation when run on a PC with i5 CPU and 4GB main memory. All the problems solvable using the lexicographic path orderings for the termination check were selected as the sample problems from [11]. For example, the problem 1 is from the group theory. It contains three equations

$$\mathcal{E}_0 = \begin{cases} f(x, f(y, z)) = f(f(x, y), z), \\ f(x, i(x)) = e, \\ f(x, e) = x, \end{cases}$$

where $\{f, i, e\}$ are function symbols (f is a binary operation, i represents the inverse and e is the identity element) and $\{x, y, z\}$ are variables. Given $\mathcal{E}_0$ and total lexicographic path orderings on $\{f, i, e\}$, the program returned a complete TRS $\mathcal{R}_c$ as follows:

$$\mathcal{R}_c = \begin{cases} f(x, i(x)) \to e \\ f(i(y), y) \to e \\ i(e) \to e \\ i(f(x, z)) \to f(i(z), i(x)) \\ i(i(x)) \to x \\ f(x, e) \to x \\ f(e, x) \to x \\ f(i(x), f(x, z)) \to z \\ f(x, f(i(x), z)) \to z \\ f(f(x, y), z) \to f(x, f(y, z)) \end{cases}$$

The computation time for each examined problem is summarized in TABLE I. The results obtained by the program using the lazy evaluation are labeled *lz-mkb*, and those obtained by the original one are labeled *mkb*. Clearly, *lz-mkb* is more efficient than *mkb* in all the problems examined.

TABLE I
COMPUTATION TIME OF MKB AND LZ-MKB

problem	mkb(ms)	lz-mkb(ms)	reduced time	reduced(%)
1	15003	1959	13044	86.94
2	160	130	30	18.75
5	14997	2738	12259	81.74
8	275	205	70	25.45
11	90	60	30	33.33
14	480	351	129	26.88
17	85	65	20	23.53
19	730	471	259	35.48
30	140	95	45	32.14
avg.	-	-	-	40.47

We have summarized the lazy values used during the experiments in TABLE II. The *label* in Node $n = \langle s : t, r_0, r_1, e\rangle$ calculates the union of $r0, r1$ and e. The *labels* in Nodes N collects all labels of the nodes in N. Associated with N is a *hash table* which stores the nodes using their size as the hash key. It is used to gain efficiency during the optional operation N.subsume(). The $size$ and $subterm$ in Term are called frequently during the whole rewrite operation.

TABLE II

Class	lazy values	
Nodes	hash table	labels
Term	size	subterm
Node	label	size(hash index)

To see the different effects to the efficiency of the program with lazy Nodes(hash table,labels), lazy Term(size,subterm) or lazy Node(label,size), we ran them separately with the same problems as TABLE I. The results are shown in TABLE III, IV, and V.

We can see the program with "Lazy Nodes Only" (TABLE III) works well with about 13 % reduced time on the average, because among all the callings of operation *union(N,N')* quite many of them return the original N, so the duplicate calculation of the hash table is avoided. Also, the examination with "Lazy Term Only" (TABLE IV) shows the best result with 26 % reduced time due to the frequency of rewriting callings during the whole procedure. However, the results with "Lazy

TABLE III
MKB AND LZ-MKB(LAZY NODES ONLY)

problem	mkb(ms)	lz-mkb(ms) lazy Nodes	reduced time	reduced(%)
1	15003	12303	2700	18.00
2	160	140	20	12.5
5	14997	14468	529	3.53
8	275	230	45	16.36
11	90	75	15	16.67
14	480	400	80	16.67
17	85	80	5	5.88
19	730	570	160	21.92
30	140	130	10	7.14
avg.	-	-	-	13.18

TABLE IV
MKB AND LZ-MKB(LAZY TERM ONLY)

problem	mkb(ms)	lz-mkb(ms) lazy Term	reduced time	reduced(%)
1	15003	2624	12379	82.51
2	160	145	15	9.38
5	14997	3890	11107	74.06
8	275	255	20	7.27
11	90	76	14	15.56
14	480	440	40	8.33
17	85	80	5	5.88
19	730	660	70	9.59
30	140	110	30	21.43
avg.	-	-	-	26.00

Node Only" (TABLE V) are not very well with only 1.2 % reduced time (they have nearly the same computation time with the program without lazy values). The label and size in Node are always called at least once for every node by Nodes to create its hash table or check the end conditions, which could be the explanation for the results in TABLE V.

V. Conclusion

We have presented *lz-mkb*: an efficient implementation of the multi-completion system MKB by using the lazy evaluation mechanism of the Scala programming language. The experiments show that *lz-mkb* is more efficient than MKB in all the problems examined. We have discussed the details by separately running the programs with different settings for the laziness. To design and implement *lz-mkb* in a physically parallel computational environment is a possible work in future. Implementation of extended versions of MKB and other algebraic reasoning systems proposed in [9] [10] [13] 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] 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.
[4] 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.

TABLE V
MKB AND LZ-MKB(LAZY NODE ONLY)

problem	mkb(ms)	lz-mkb(ms) lazy Node	reduced time	reduced(%)
1	15003	14880	123	0.82
2	160	155	5	3.13
5	14997	14921	76	0.51
8	275	270	5	1.82
11	90	90	0	0
14	480	475	5	1.04
17	85	85	0	0
19	730	725	5	0.68
30	140	136	4	2.86
avg.	-	-	-	1.21

[5] D. E. Knuth and P. B. Bendix, "Simple word problems in universal algebras," *J. Leech(ed.), Computational Problems in Abstract Algebra*, Pergamon Press, 1970, pp.263-297.
[6] M. Kurihara and H. Kondo, "Completion for multiple reduction orderings," *Journal of Automated Reasoning*, Vol.23, No.1, 1999, pp.25-42.
[7] M. Odersky, *Programming in Scala,* 2nd ed., Artima Press, 2010.
[8] 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.
[9] 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
[10] H. Sato, M. Kurihara, S. Winkler, A. Middeldorp, "Constraint-Based Multi-Completion Procedures for Term Rewriting Systems," *IEICE Transactions on Information and Systems*, Vol. E92.D, No.2, 2009, pp.220-234
[11] J. Steinbach and U. Kühler, "Check your ordering - termination proofs and open problems," SEKI report SR-90-25 (SFB), Fachbereich Informatik, Universität Kaiserslautern, Germany, 1990.
[12] Terese, *Term rewriting systems.* Cambridge University Press. 2003.
[13] S. Winkler, H. Sato, A. Middeldorp, M. Kurihara, "Multi-Completion with Termination Tools," *Journal of Automated Reasoning*, Vol.50, Issue 3, March 2013, pp.317-354.