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, calledlz-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 thatlz-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 equivalently to the input set of equations. The resultant TRS can be used to decide the equational consequences (word problems) of the input equations.

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 → t, t → 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, calledlz-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 developlz-mkb. The actual implementation oflz-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 implemen-
inition of $\text{Iz-mkb}$. 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 $\Sigma$ 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 $\Sigma$ with arity $n=0$ are called constant symbols.

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

Definition 2.2: The set $T(\Sigma, V)$ of all terms over $\Sigma$ 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 $\sigma : V \rightarrow T(\Sigma, V)$ such that $\sigma(x) \neq x$ for only finitely many $x$s. We can extend any substitution $\sigma$ to a mapping $\sigma : T(\Sigma, V) \rightarrow T(\Sigma, V)$ by defining $\sigma(f(s_1, \ldots, s_n)) = f(\sigma(s_1), \ldots, \sigma(s_n))$. The application $\sigma(s)$ of $\sigma$ to $s$ is often written as $s \sigma$. A term $t$ is an instance of a term $s$ if there exists a substitution $\sigma$ such that $s \equiv \sigma t$. Two terms $s$ and $t$ are variants of each other and denoted by $s \equiv \sigma t$, 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 $l \rightarrow r$ 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 $\emptyset$ be a new symbol which does not occur in $\Sigma \cup V$. A context, denoted by $C$, is a term $t \in T(\Sigma, V \cup \{\emptyset\})$ with exactly one occurrence of $\emptyset$. $C[s]$ denotes the term obtained by replacing $\emptyset$ in $C$ with $s$.

Definition 2.4: The reduction relation $\rightarrow_R \subseteq T(\Sigma, V) \times T(\Sigma, V)$ is defined by $s \rightarrow_R r$ if there exists a rule $l \rightarrow r$ in $R$, a context $C$, and a substitution $\sigma$ such that $s \equiv C[l \sigma]$ and $C[r \sigma] \equiv t$. A term $s$ is reducible if $s \rightarrow_R r$ for some $t$; otherwise, $s$ is a normal form.

A TRS $R$ terminates if there is no infinite rewrite sequence $s_0 \rightarrow_R s_1 \rightarrow_R \ldots$. 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 $\sigma$.
- closed under context: $s \succ t$ implies $C[s \sigma] \succ C[t \sigma]$ for any context $C$.
- well-founded: if there exist no infinite decreasing sequences $t_1 \succ t_2 \succ t_3 \succ \ldots$.

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

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 $s \sqsubseteq t$), if there exists a term $v$ such that $s \rightarrow_R v$ and $t \rightarrow_R v$, and where $\rightarrow_R$ is the reflexive transitive closure of $\rightarrow_R$.

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

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

With Definition 2.9 we can define critical pairs as follows:

Definition 2.10: Consider two rewrite rules $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$ 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 $\text{mgu}(s, l_2) = \sigma$, then the pair $(l_1[s \sigma], r_1[s \sigma])$ of terms is called a critical pair of $l_1 \rightarrow r_1$ and $l_2 \rightarrow r_2$.

For example, let $f$ be a function symbol. $\{a, b, c\}$ be variables, and consider two rewrite rules $f(a) \rightarrow b$ and $a \rightarrow c$. 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 $\text{mgu}$ (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 $(f(c), b)$ 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 $p \sqsubseteq q$.

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 $\mathcal{R}_0$ 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: $(\mathcal{E} \cup \{s \rightarrow s\}; \mathcal{R}) \vdash (\mathcal{E}; \mathcal{R})$
COMPOSE: \((E; R \cup \{s \rightarrow t\}) \vdash (E; R \cup \{s \rightarrow u\})\)
if \(t \rightarrow_R u\)

SIMPLIFY: \((E \cup \{s \rightarrow t\}; R) \vdash (E \cup \{s \rightarrow u\}; R)\)
if \(t \rightarrow_R u\)

ORIENT: \((E \cup \{s \rightarrow t\}; R) \vdash (E; R \cup \{s \rightarrow t\})\)
if \(s \rightarrow t\)

COLLAPSE: \((E; R \cup \{t \rightarrow s\}) \vdash (E \cup \{u \rightarrow s\}; R)\)
if \(l \rightarrow_R c \in R, t \rightarrow_{(t \rightarrow s)} u, \) and \(t \rightarrow l\)

DEDUCE: \((E; R) \vdash (E \cup \{s \rightarrow t\}; R)\)
if \(u \rightarrow_R s\) and \(u \rightarrow_R t\)

The new symbol \(\triangleright\) here denotes the encompassment ordering defined as follows.

**Definition 2.12:** An encompassment order \(\triangleright\) on a set of terms is defined by \(s \triangleright t\) if some subterm of \(s\) is an instance of \(t\) and \(s \neq t\).

For example, if \(\{f, g\}\) are function symbols and \(\{x, y, z\}\) variables, then \(f(x, g(x)) \triangleright (y, g(z))\) but \(f(x, g(y)) \not\triangleright f(z, g(z))\).

**Definition 2.13:** A node \((s : t, R_0, R_1, E)\), where \(s : t\) is an ordered pair of terms \(s\) and \(t\) called datum, and \(R_0, R_1, E\) are subsets of \(I\) labeled rules such that:

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

Intuitively, the set \(R_0(R_1)\) represents the indices of processes executing KB in which the set of rewrite rules \(R\) currently contains \(s \rightarrow t\) (\(t \rightarrow s\)), and \(E\) represents those of \(E\) processes, which contains an equation \(s \rightarrow t\) (\(t \rightarrow s\)).

The node \((s : t, R_0, R_1, E)\) is considered to be identical with the node \((t : s, R_1, R_0, E)\), hence the inference rules of MKB working on a set \(N\) of nodes defined below implicitly specify the symmetric cases.

**DELETE:** \(N \cup \{\{s : s, 0, 0, E\}\} \vdash N\)
if \(E = \emptyset\)

**ORIENT:** \(N \cup \{\{s : t, R_0, R_1, E \cup E'\}\} \vdash N \cup \{\{s : t, R_0 \cup E', R_1, E\}\}\)
if \(E' = \emptyset, E \cap E' = \emptyset,\) and \(s \triangleright_i t\) for all \(i \in E'\)

**REWITE_1:** \(N \cup \{\{s : t, R_0, R_1, E\}\} \vdash N \cup \{\{s : t, R_0 \setminus R, R_1, E' \setminus R\}\}\)

**REWITE_2:** \(N \cup \{\{s : t, R_0, R_1, E\}\} \vdash N \cup \{\{s : t, R_0 \cap R, R_1 \cap E, R\}\}\)
if \(\{l : r, R, \ldots\} \in N, t \rightarrow_{(t \rightarrow E)} u, \) and \(R_0 \cap E \cap R = \emptyset\)

**SUBSUME:** \(N \cup \{\{s : t, R_0, R_1, E\}\} \vdash N \cup \{\{s : t, R_0 \cap R_1, E', E'\}\}\)
if \(s : t \rightarrow E\)

**GC:** \(N \cup \{\{s : t, 0, 0, 0\}\} \vdash N\)

**III. IMPLEMENTATION**

In this section we will discuss the details about the implementation.

We implemented an algebraic reasoning system called \(Iz-mkb\) 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, substitutions, 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 \textit{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 \textit{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 \texttt{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 \texttt{REWRITE} and \texttt{DEDUCE}.

The operation \texttt{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 \texttt{rewritten}() is not included in the class of nodes but it takes nodes as arguments. In the original idea of MKB, \texttt{REWRITE-1} and \texttt{REWRITE-2} simulate the \texttt{COMPOSE, SIMPLIFY} and \texttt{COLLAPSE} (if appropriate conditions are satisfied) in one single operation. More exactly, \texttt{REWRITE-1} and \texttt{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 \((D, N, M)\) where:

\begin{itemize}
  \item \(D\): the nodes rewritten by \texttt{rewritten}() (i.e., the original ones with the original datum \(s:t\))
  \item \(N\): the nodes “created” by \texttt{rewritten}() (i.e., the new nodes with the original datum \(s:t\) and updated labels)
  \item \(M\): the nodes “modified” during \texttt{rewritten}() (i.e., the new nodes with a new datum \(s:u\) and updated labels)
\end{itemize}

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

Notice that to the symmetric cases of nodes, we just use the \texttt{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 \(\{\langle N, N'\rangle, \langle N, n', N\rangle, \langle n, N, N'\rangle, \langle n, N', N\rangle\}\) one by one. Surely \((N, N')\) is updated after every single \texttt{rewrite-1} or \texttt{rewrite-2}. In this way, we obtain a tuple \((D, N, M)\) of three nodes in which every calculated node is included and no more rewrite can be applied. Finally, the tuple \((D, N, M)\) is returned as the result of the operation \texttt{rewritten}().

The operation \texttt{N.deduce()} 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 \(n = \{a:b, R_0, R_1\ldots\}\) and \(n' = \{c:d, R_0', R_1'\ldots\}\). 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 \(\{R_0 \cap R_0', R_1 \cap R_1'\ldots\}\).

The operation \texttt{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 \texttt{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 \textit{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 \texttt{N.subsume()} is invoked by the operation \texttt{union(N, N')} which is designed for combining nodes \(N\) and \(N'\). We observe that in every iteration of the while loop, \texttt{union(N, N')} operation is called at least once (i.e., for every chosen \(n\), \texttt{subsume()} would be called at line 9 once; And for those that satisfied the proper conditions of line 11 and line 13, two more operations are required). This means \texttt{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 \([s, N]\), where \(N\) is a \texttt{list} of nodes and \(J_n\) is a lazy value defined in the node class as the \texttt{size} of the node (i.e., for a node \(n = \{s : t, r_0, r_1, c\}\), \texttt{size} = \texttt{s.size} + \texttt{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 size as its key. In other words, for every \(n \in N\), \(n\) uses its \texttt{size} \(J_n\) as the key to \([s, N]\), then the set \(N_i\) containing all the nodes with same \texttt{size} \(J_n\) is looked up for the nodes with variant data. In our Scala program, the hash map \([s, N]\) is declared to be \texttt{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 \texttt{success()} 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 \(N_v\) nodes. Then \(E[N_o \cup N_v, i] = \emptyset\) and \(\exists [N_v, i] \) is a convergent set of rewrite rules contained in \(\geq \). We also created lazy values in nodes.

\(^1\)a data structure defined in Scala’s library

(Advance online publication: 10 July 2015)
Algorithm 1 \( \text{lz-mkb}(E, O) \)

1: \( N_0 := \{(s : t, \emptyset, \emptyset, I) \mid s \leftrightarrow t \in E\} \) where \( I = \{1, \ldots, |O|\} \)
2: \( N_c := \emptyset \)
3: while success\( (N_o, N_c) = \text{false} \) do
4: if \( N_o = \emptyset \) then
5: return(fail)
6: else
7: \( n := N_o.\text{choose}() \)
8: \( (D, N, M) := \text{rewrite}(\{n\}, N_o) \)
9: \( N_o := \text{union}(N_o - \{n\}, \text{N.delete}()) \)
10: \( n := M.\text{head} \)
11: if \( n \neq (\ldots, \emptyset, \emptyset, \ldots) \) then
12: \( n := n.\text{orient}() \)
13: if \( n \neq (\ldots, \emptyset, \emptyset, \ldots) \) then
14: \( (D, N, M) := \text{rewrite}(N_c, \{n\}) \)
15: \( N_o := \text{union}(N_o, \text{N.delete}()) \)
16: \( N_c := N_c + M - D \)
17: \( N_c := \text{N_c.garbagecollect}() \)
18: \( N_o := \text{union}(N_o, \text{deduce}(n, N_c).\text{delete}()) \)
19: end if
20: \( N_c := \text{union}(N_c, \{n\}) \)
21: end if
22: end if
23: end while
24: return \( R_c[N_c, i] \) where \( i = \text{success}(N_o, N_c) \)

Liquid cache (hash table) was used to hold the occurrences of the index \( i \) in the labels, so that we do not need to calculate it in the unchanged \( N_o \) every time. This also makes the computation efficient as \( N.\text{choose}() \) 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

\[
\mathcal{E}_0 = \left\{ \begin{array}{l}
  f(x, f(y, z)) = f(f(x, y), z), \\
  f(x, i(x)) = x, \\
  f(x, e) = x,
\end{array} \right. \]

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

\[
\mathcal{R}_c \text{ as follows:} \quad \\
\mathcal{R}_c = \left\{ \begin{array}{l}
  f(x, i(x)) \rightarrow e, \\
  f(i(y), y) \rightarrow e, \\
  i(e) \rightarrow e, \\
  i(f(x, z)) \rightarrow f(i(z), i(x)), \\
  i(i(x)) \rightarrow x, \\
  f(x, e) \rightarrow x, \\
  f(x, f(x, z)) \rightarrow z, \\
  f(x, f(i(x), z)) \rightarrow z, \\
  f(x, f(y, z)) \rightarrow f(x, f(y, z)).
\end{array} \right. \]

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

<table>
<thead>
<tr>
<th>problem</th>
<th>( \text{mkb}(\text{ms}) )</th>
<th>( \text{lz-mkb}(\text{ms}) )</th>
<th>reduced time</th>
<th>reduced(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>15003</td>
<td>1939</td>
<td>13044</td>
<td>86.94</td>
</tr>
<tr>
<td>2</td>
<td>160</td>
<td>130</td>
<td>30</td>
<td>18.75</td>
</tr>
<tr>
<td>5</td>
<td>14997</td>
<td>2738</td>
<td>12259</td>
<td>81.74</td>
</tr>
<tr>
<td>8</td>
<td>275</td>
<td>205</td>
<td>70</td>
<td>25.45</td>
</tr>
<tr>
<td>11</td>
<td>90</td>
<td>60</td>
<td>30</td>
<td>33.33</td>
</tr>
<tr>
<td>14</td>
<td>480</td>
<td>351</td>
<td>129</td>
<td>26.88</td>
</tr>
<tr>
<td>17</td>
<td>85</td>
<td>65</td>
<td>20</td>
<td>23.53</td>
</tr>
<tr>
<td>19</td>
<td>730</td>
<td>471</td>
<td>239</td>
<td>35.48</td>
</tr>
<tr>
<td>30</td>
<td>140</td>
<td>95</td>
<td>45</td>
<td>32.14</td>
</tr>
<tr>
<td>avg</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>40.47</td>
</tr>
</tbody>
</table>

We have summarized the lazy values used during the experiments in TABLE II. The label in Node \( n = (s : t, r_0, r_1, e) \) calculates the union of \( r_0, r_1 \) 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 \( N.\text{subsume}() \). The size and subterm in Term are called frequently during the whole rewrite operation.

<table>
<thead>
<tr>
<th>Class</th>
<th>lazy values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nodes</td>
<td>hash table</td>
</tr>
<tr>
<td>Term</td>
<td>size</td>
</tr>
<tr>
<td>Node</td>
<td>label</td>
</tr>
</tbody>
</table>

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 \( \text{union}(N, N') \) 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

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

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


TABLE V

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