An Efficient Implementation of Multi-Context Algebraic Reasoning System with Lazy Evaluation

ChengCheng Ji, Haruhiko Sato and Masahito Kurihara

Abstract—A multi-context algebraic reasoning system is a computational system which can efficiently simulate 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. MKB handles multiple contexts each corresponding to a reduction ordering, and normally, one of them leads to a success and others cause the failure or the divergence. In this study, we present an efficient implementation of MKB, called lz-mkb, which exploits the lazy evaluation mechanism of a functional, object-oriented programming language Scala. The experiment shows that lz-mkb is more efficient than the original MKB implementation of Kurihara and Kondo.

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

I. INTRODUCTION

ALGEBRAIC computation systems such as term rewriting systems (TRSs) play a fundamental role in various areas of computer science, including automated theorem proving, analysis and implementation of abstract data types, and decidability of word problems [1]. In applications, two properties of term rewriting systems called termination and confluence are often required. A TRS is complete (or convergent) if it satisfies termination and confluence at the same time.

Knuth and Bendix have proposed a standard completion procedure called KB to generate a complete TRS [1]. Given a set of equations and a reduction ordering on a set of terms, the KB 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 term rewriting system equivalently to the input set of equations. Such a system can be used to decide the equational consequences (word problems) of the input equations. The KB leads to three possible results: success, failure, or divergence. In the success case, the procedure stops and outputs a complete set of rewrite rules. In the failure case, the procedure stops but returns a partial result with unorientable equations. 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 orderings. 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.

This problem was partially solved by a completion procedure called MBK [2]. MBK is a single procedure that efficiently simulates execution of multiple processes each running KB with a different reduction ordering. The key idea of MBK is 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 $s \rightarrow t$, $t \rightarrow s$, or an equation $s = t$. The well-designed inference rules of MBK allows an efficient simulation of multiple inferences in several processes all in a single operation.

Scala is a rising programming language supporting both functional programming and object-oriented programming. It provides a feature called lazy evaluation which can increase performance of programs by avoiding needless calculations. In this study, we present an efficient implementation of MBK, called lz-mkb, which exploits the lazy evaluation mechanism of Scala.

This paper is organized as follows. In Section 2, we will have a brief review on term rewriting systems and completion procedures KB and MBK. In Section 3, we will discuss the implementation of lz-mkb. The result of the experiments will be shown in Section 4. In Section 5, 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) [5] [6] [7] [8] [9]. 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 $n \neq 0$ 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 $s = t$ when the terms $s$ and $t$ are identical. A term $s$ is a subterm
of $t$, if either $s \equiv t$ or $t \equiv (t_1, \ldots, t_n)$ 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 $\sigma(f(s_1, \ldots, s_m)) = f(\sigma(s_1), \ldots, \sigma(s_m))$. 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 $s\sigma \equiv t$. Two terms $s$ and $t$ are variants of each other and denoted by $s \equiv t$, 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 $\Box$ 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 $\rightarrow_R \subseteq T(\Sigma, V) \times T(\Sigma, V)$ is defined by $s \rightarrow_R t$ iff 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 $s \rightarrow_R t$ for some $t$; otherwise, $s$ is a normal form.

A TRS $R$ terminates if there is no infinite rewrite sequence $s_0 \rightarrow_R s_1 \rightarrow_R \cdots$. 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 $\sigma$.
- closed under context: $s \succ t$ implies $C[s] \succ C[t]$ for any context $C$.
- well-founded: there exist no infinite decreasing sequences $l_1 \succ l_2 \succ l_3 \succ \cdots$.

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

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

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

**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 is a substitution $\sigma$ such that $s\sigma \equiv t\sigma$, $\sigma$ 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 $l_1 \rightarrow r_1$ 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 $mgu(s, l_2) = \sigma$, then the pair $(l_1[\sigma][r_2], r_1[\sigma])$ 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 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 $p \downarrow q$. 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 [1] [2].

Given a set of equations $E_0$ 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 $E_0$. The KB procedure implements the following six inference rules.

**DELETE:** $(E \cup \{s \leftarrow t; \mathcal{R}\} \vdash \langle E; \mathcal{R} \rangle)$ if $t \rightarrow_\mathcal{R} u$

**COMPOSE:** $(E; \mathcal{R} \cup \{s \leftarrow t\}) \vdash \langle E; \mathcal{R} \cup \{s \leftarrow u\} \rangle$ if $t \rightarrow_\mathcal{R} u$

**SIMPLIFY:** $(E \cup \{s \leftarrow t; \mathcal{R}\} \vdash \langle E \cup \{s \leftarrow u; \mathcal{R}\} \rangle$ if $s \succ t$

**ORIENT:** $(E; \mathcal{R} \cup \{t \leftarrow s\}) \vdash \langle E; \mathcal{R} \cup \{u \leftarrow s; \mathcal{R}\} \rangle$ if $t \rightarrow_\mathcal{R} u$, and $t \succ l$

**COLLAPSE:** $(E; \mathcal{R}) \vdash \langle E \cup \{s \leftarrow t; \mathcal{R}\} \rangle$ if $u \rightarrow_\mathcal{R} s$ and $u \rightarrow_\mathcal{R} t$

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

**Definition 2.12:** An encompassment ordering $\triangleright$ on a set of terms is defined by $s \triangleright t$ iff 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 $\{f(x, g(x)) \triangleright f(y, g(z)) \}$ but $f(x, g(y)) \not\triangleright f(z, g(z))$. KB starts from the initial state $(E_0, \mathcal{R}_0)$ where $\mathcal{R}_0 = \emptyset$. The procedure changes the states in a possibly infinite completion sequence $(E_0; \mathcal{R}_0) \vdash \langle E_1; \mathcal{R}_1 \rangle \vdash \cdots$ by its inference rules. The result of the completion sequence is the sets $E_\infty$ and $\mathcal{R}_\infty$. When $E_\infty = \emptyset$, $\mathcal{R}_\infty$ will be a confluent and terminating TRS satisfying $\rightarrow_\mathcal{R}_\infty = \rightarrow_\mathcal{R}_\infty^{\mathcal{R}_\infty}$, which means KB procedure has succeeded. And the sequence has failed if $E_\infty \neq \emptyset$.

A completion procedure for multiple reduction orderings called MKB developed in [2] 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 labels such that:

- \( R_0, R_1 \) and \( E \) are mutually disjoint, (i.e., \( R_0 \cap R_1 \) is a single operation is based on a special data structure called

- \( i \in R_0 \) implies \( s \succ_i t \), and \( i \in R_1 \) implies \( t \succ_i s \)

Intuitively, the set \( R_0 \cap R_1 \) represents the indices of processes executing KB in which the set \( \mathcal{R} \) currently contains \( s \rightarrow t \) (\( t \rightarrow s \)), and \( E \) represents those of processes in which \( E \) contains an equation \( s \rightarrow t \) (\( t \rightarrow s \)). 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:**
\[
N \cup \{ \langle s : t, 0, 0, E \rangle \} \vdash N
\]
if \( E \neq \emptyset \)

**ORIENT:**
\[
N \cup \{ \langle s : t, R_0, R_1, E \cap E' \rangle \} \vdash N \cup \{ \langle s : t, R_0 \cap R_1, E \rangle \}
\]
if \( E' \neq \emptyset \), \( E \cap E' = \emptyset \), and \( s \succ t \) for all \( i \in E' \)

**REWRITE 1:**
\[
N \cup \{ \langle s : t, R_0, R_1, E \rangle \} \vdash N \cup \{ \langle s : u, R_0 \cap R_1, R_1 \rangle \}
\]
if \( \{ i : r, \ldots, r_i \} \in N, t \rightarrow_{(s \rightarrow t)} u, t \equiv t, \) and \( \{ R_0 \cap R_1, R_1 \} \)

**REWRITE 2:**
\[
N \cup \{ \langle s : t, R_0, R_1, E \rangle \} \vdash N \cup \{ \langle s : u, R_0 \cap R_1, E \rangle \}
\]
if \( \{ i : r, \ldots, r_i \} \in N, t \rightarrow_{(s \rightarrow t)} u, t \equiv t, \) and \( \{ R_0 \cup R_1, R_1 \} \)

**DEDUCE:**
\[
N \vdash N \cup \{ \langle s : t, 0, 0, E \rangle \}
\]
if \( \{ i : r, \ldots, r_i \} \in N, t \equiv t, \) and \( \{ E \cap E' \} \)

**GC:**
\[
N \cup \{ \langle s : t, R_0, R_1, E \rangle \} \vdash N
\]
if \( \{ s : t, R_0, R_1, E \} \)

**SUBSUME:**
\[
N \cup \{ \langle s : t, R_0, R_1, E \rangle \} \vdash N \cup \{ \langle s' : t', R_0, R_1, E' \rangle \}
\]
if \( s : t \) and \( s' : t' \) are variants and \( E'' \) is a tuple

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 \( n = \{ s : t, R_0, R_1, E \} \) 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
\[
\mathcal{E}[n,i] = \begin{cases} 
\{ s \rightarrow t \}, & \text{if } i \in E, \\
\emptyset, & \text{otherwise.}
\end{cases}
\]

Similarly, the \( \mathcal{R} \)-projection \( \mathcal{R}[n,i] \) of \( n \) onto \( i \) is a set of rules defined by
\[
\mathcal{R}[n,i] = \begin{cases} 
\{ s \rightarrow t \}, & \text{if } i \in R_0, \\
\{ t \rightarrow s \}, & \text{if } i \in R_1, \\
\emptyset, & \text{otherwise.}
\end{cases}
\]

These notions can also be extended for a set \( N \) of nodes as follows:
\[
\mathcal{E}[N,i] = \bigcup_{n \in N} \mathcal{E}[n,i], \quad \mathcal{R}[N,i] = \bigcup_{n \in N} \mathcal{R}[n,i]
\]

MKB starts with the initial set \( N_0 \) of nodes:
\[
N_0 = \{ \langle s : t, 0, 0, I \rangle \mid s \approx t \in \mathcal{E}_0 \}
\]
which means, given the initial set of equations \( \mathcal{E}_0 \), we have \( \{ E[N_0,i]; R[N_0,i] \} = \{ E_0; \emptyset \} \) for all \( i \in I \). The state sequence of MKB is generated as \( N_0 \vdash N_1 \vdash \cdots \vdash N_c \).

If \( E[N_c,i] \) is empty and all critical pairs of \( R[N_c,i] \) have been created, MKB returns \( R[N_c,i] \) 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 \( \text{lz-mkb} \) based on MKB in [2] by using lazy evaluation mechanism of the programming language Scala. Scala is a programming language which possesses the abilities supporting 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 philosophy of functional programming (e.g., “uniform return type” principle [3]) 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 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 \text{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 rewrite \text{REWRITE} and \text{DEDUCE}.

The operation \text{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

\footnote{A data structure defined in Scala’s library}
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 \(\text{rewrite}(N, N')\) is not included in the class
of nodes but it takes nodes as arguments. In the original
idea of MKB, \text{REWRITE}_1 and \text{REWRITE}_2 simulate the
\text{COMPOSE}, \text{SIMPLIFY} and \text{COLLAPSE} (if appropriate
conditions are satisfied) in one single operation. More
exactly, \text{REWRITE}_1 and \text{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

\[
N := N \setminus \{\text{original nodes}\} \cup \{\text{updated nodes}\}
\]

In our implementation, we follow the discipline of func-
tional 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:

\[
D: \quad \text{the nodes rewritten by rewrite}(N, N')\text{(i.e., the original}
\quad \text{ones with the original datum } s : t)
\]

\[
N: \quad \text{the nodes “created” by rewrite}(N, N')\text{(i.e., the new}
\quad \text{nodes with the original datum } s : t \text{ and updated}
\quad \text{labels)}
\]

\[
M: \quad \text{the nodes “modified” during rewrite}(N, N')\text{(i.e., the}
\quad \text{new nodes with a new datum } s : u \text{ and updated}
\quad \text{labels)}
\]

Notice that to the symmetric cases of nodes, we
just use the \textit{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
\text{rewrite}_1 or \text{rewrite}_2. In this way, we obtain a tuple
\((D_{\infty}, N_{\infty}, M_{\infty})\) of three nodes in which every
calculated node is included and no more rewrite can be applied.
Finally, the tuple \((D_{\infty}, N_{\infty} - D_{\infty}, M_{\infty} - D_{\infty})\)
is returned as the result of the operation \text{rewrite}(N, N').

The operation \(N.deduce(n)\) generates all the possible
critcal 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 \(\{a \iff b, c \iff d\}\), which means the modification
of labels should be considered for each of \(R_0 \cap R'_0, R_1 \cap
R'_1, \ldots\) and \(R_0 \cap R'_1, R'_1 \cap R'_0\).

The operation \(N.garbagecollect()\) has no related infer-
ence 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 \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 [2]. The operation \(N.subsume()\)
is invoked by the operation \text{union}(N, N') which is designed
for combining nodes \(N\) and \(N'\). We observe that in every
iteration of the while loop, \text{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 ones satisfied the
proper conditions of line 11 and line 13, two more operations
are required). This means the \text{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 \(\mathcal{J}[s, N]\), where \(N\) is a list
of nodes and \(\mathcal{J}\) is a lazy value defined in the node class as
the size of the node (i.e., for a node \(n = \{s : t, r_0, r_1, e\}\),
the size = \((s : t).size + r_0.size + r_1.size + e.size)\), so that
we need only check the nodes corresponding to the original
ones’ sizes by using the hash map as indices at one time.
In other words, for every \(n \in N\), \(n\) uses its size \(\mathcal{J}_n\) as the
key to \(\mathcal{J}[s, N]\), then the set \(N_0\) containing all the nodes with
same size \(\mathcal{J}_n\) is used for searching the nodes with variant
data. The hash map \(\mathcal{J}[s, N]\) was announced by lazy means it
was calculated only at the first time, then it would be stored
as a constant preparing for the callings since then.

\begin{algorithm}
1: \(N_0 := \{(s : t, \emptyset, \emptyset, I) \mid s \leftrightarrow t \in E\} \text{ where } I = \{1, \ldots, |O|\}\)
2: \(N_c := \emptyset\)
3: while success\((N_0, N_c) = false\) do
4: \(\text{if } N_0 = \emptyset \text{ then}
5: \text{return} \text{fail}\)
6: \text{else}
7: \(n := N_0.\text{choose}()\)
8: \(k := \text{rewrite}\((\{n\}, N_c)\)\)
9: \(N_0 := \text{union}(N_0 - \{n\}, k_2.\text{delete}())\)
10: \(n := k_3.\text{head}\)
11: \(\text{if } n \neq \{\ldots, \emptyset, \emptyset, \emptyset\} \text{ then}
12: \(n := n.\text{orient}()\)
13: \(\text{if } n \neq \{\ldots, \emptyset, \emptyset, \ldots\} \text{ then}
14: \(j := \text{rewrite}(n, \{n\})\)\)
15: \(N_0 := \text{union}(N_0, j_2.\text{delete}())\)
16: \(N_c := N_c + j_3 - J_1\)
17: \(N_c := N_c.\text{garbagecollect}()\)
18: \(N_0 := \text{union}(N_0, \text{deduce}(n, N_c).\text{delete}())\)
19: end if
20: \(N_c := \text{union}(N_c, \{n\})\)
21: end if
22: end if
23: end while
24: return \(\mathcal{R}[N_c, i] \text{ where } i = \text{success}(N_0, N_c)\)
\end{algorithm}

Notice that the procedure success\((N_0, N_c)\) 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 \(N_0\) and any \(E\) labels of \(N_c\) nodes. Then \(E[N_0 \cup N_c, i] = \emptyset\) and \(\mathcal{R}[N_c, i]\) is a convergent set of rewrite
rules contained in \(\succ\). We also created lazy values in nodes
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.

Algorithm 1 lz-mkb(E, O)
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 Pentium 4 CPU and 2 GB main memory. The sample problems are taken from [4]. For example, the problem 3.01 is from the group theory. It contains three equations

$$e_0 \begin{cases} f(x, f(y, z)) = f(f(x, y), z), \\ f(x, i(x)) = e, \\ f(x, e) = x, \end{cases}$$

where $\{f, i, e\}$ are function symbols ($f$ is a binary operation, $i$ represents inverse and $e$ is the identity element) and $\{x, y, z\}$ are variables. Given $e_0$ and total lexicographic path orderings on $\{f, i, e\}$, the program would return a complete TRS $R_e$ as follows: ²

$$R_e = \begin{cases} f(x, i(x)) \rightarrow e, \\ f(i(y3), y3) \rightarrow e, \\ i(e) \rightarrow e, \\ i(f(x20, z41)) \rightarrow f(i(z41), i(x20)) \\ i(i(x1)) \rightarrow x1, \\ f(x, e) \rightarrow x, \\ f(e, x5) \rightarrow x5, \\ f(i(x3), f(x3, z)) \rightarrow z, \\ f(x1, f(i(x1), z)) \rightarrow z, \\ f(f(x, y), z) \rightarrow f(x, f(y, z)) \end{cases}$$

The computation time of the examined problems are summarized in Table 1 as results. 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>
<thead>
<tr>
<th>problem</th>
<th>3.01</th>
<th>3.08</th>
<th>3.10</th>
<th>3.24</th>
<th>peano</th>
<th>collapse</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkb(ms)</td>
<td>13491</td>
<td>213</td>
<td>94</td>
<td>139</td>
<td>251</td>
<td>169</td>
</tr>
<tr>
<td>lz-mkb(ms)</td>
<td>11273</td>
<td>282</td>
<td>62</td>
<td>109</td>
<td>188</td>
<td>94</td>
</tr>
</tbody>
</table>

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. 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 [10] [11] [12] is also an interesting future work.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Number 25330074.

²The variables with postfix numbers are the renamed variables.

REFERENCES