IEICE TRANS. INF. & SYST., VOL.E92–D, NO.2 FEBRUARY 2009

PAPER Special Section on Foundations of Computer Science

Constraint-Based Multi-Completion Procedures for Term Rewriting Systems

Haruhiko SATO[†a)], ***Student Member***, **Masahito KURIHARA**[†b)], ***Member***, **Sarah WINKLER**[††c)], ***and*** **Aart MIDDELDORP**[††d)], ***Nonmembers***

SUMMARY In equational theorem proving, convergent term rewriting systems play a crucial role. In order to compute convergent term rewriting systems, the standard completion procedure (KB) was proposed by Knuth and Bendix and has been improved in various ways. The *multi-completion* system MKB developed by Kurihara and Kondo accepts as input a set of reduction orders in addition to equations and efficiently simulates parallel processes each of which executes the KB procedure with one of the given orderings. Wehrman and Stump also developed a new variant of completion procedure, *constraint-based completion*, in which reduction orders need not be given by using automated modern termination checkers. As a result, the constraint-based procedures simulate the execution of parallel KB processes in a sequential way, but naive search algorithms sometimes cause serious inefficiency when the number of the potential reduction orders is large. In this paper, we present a new procedure, called a *constraint-based multi-completion* procedure MKBcs, by augmenting the constraint-based completion with the framework of the multi-completion for suppressing the combinatorial explosion by sharing inferences among the processes. The existing constraint-based system Slothrop, which basically employs the best-first search, is more efficient when its built-in heuristics for process selection are appropriate, but when they are not, our system is more efficient. Therefore, both systems have their role to play.

key words: *equational theorem proving, term rewriting system, Knuth-Bendix completion, multi-completion, constraint-based multi-completion*

1. Introduction

Term rewriting systems [2], [4], [8], [16] play an important role in various areas, such as automated theorem proving, functional and logic programming languages, and algebraic specification of abstract data types. In many applications, termination and confluence are crucially important properties of term rewriting systems. A term rewriting system which has both of these properties is said to be convergent.

In order to compute a convergent term rewriting system, the *standard completion* procedure (KB) was proposed by Knuth and Bendix [9] and has been improved in various ways [3]. Given a set of equations $\mathcal{E}_0$ (or rewrite rules $\mathcal{R}_0$) and a reduction order on a set of terms, the procedure tries to generate a convergent term rewriting system which is equationally equivalent to $\mathcal{E}_0$ (or $\mathcal{R}_0$) by adding or modifying rewrite rules. The reduction orders are used for orienting the equations (either from left to right or from right to left) in order to ensure the termination of the resultant systems. The success of the procedure heavily depends on the choice of the reduction order to be supplied. Such a choice is often difficult for general users to make unless they have good insight in termination proof techniques. Unfortunately, one cannot try out potentially-appropriate reduction orders one by one (*sequentially*), because one of those runs may lead to infinite, divergent computation and prohibit the exploration of the remaining possibilities.

Kurihara and Kondo [10] partially solved this problem by developing a completion procedure called MKB, which, accepting as input a *set* of reduction orders as well as equations, efficiently simulates (in a single process) *parallel* execution of KB procedures each working with one of those orders. The key idea is the development of the data structure for storing a pair $s : t$ of terms associated with the information to show which processes contain the rule $s \to t$ (or $t \to s$) and which processes contain the equation $s \approx t$. This structure makes it possible to define a meta-inference system for MKB that effectively simulates a lot of closely-related inferences made in different processes all in a single operation. We call this type of procedure a *multi-completion* procedure.

As another approach to this problem, Wehrman, Stump and Westbrook [18] developed a new procedure in which no orders need to be provided by the users. The idea is that the procedure keeps constraints (a set of rewrite rules) on reduction orders and checks the existence of a reduction order satisfying those constraints by using an external automated termination checker. Using the state-of-the-art, modern termination checkers, the procedure can be virtually supplied with the richest family of mechanically-checkable reduction orders and can solve the widest variety of completion problems. This is true at least theoretically, but in reality, there is an inefficiency problem caused by the combinatorial explosion. Unlike the standard completion, the constraint-based procedures should orient the equations in both directions (for example, by using the breadth-first search) in order to ensure the completeness of the search algorithm. This often causes the exponential increase in the number of reduction orders before creating the solution.

In this paper, we present a new multi-completion procedure MKBcs by combining MKB with the constraint-

Manuscript received April 1, 2008.
Manuscript revised July 2, 2008.

[†]The authors are with the Graduate School of Information Science and Technology, Hokkaido University, Sapporo-shi, 060–0814 Japan.
[††]The authors are with the Institute of Computer Science, University of Innsbruck, Austria.
a) E-mail: haru@complex.eng.hokudai.ac.jp
b) E-mail: kurihara@ist.hokudai.ac.jp
c) E-mail: Sarah.Winkler@uibk.ac.at
d) E-mail: Aart.Middeldorp@uibk.ac.at

DOI: 10.1587/transinf.E92.D.220

based completion procedures. The inference rules of MKBcs are almost the same as MKB, but the semantics of "processes" are totally different and we have had to develop a special scheme for encoding the indexes of the dynamically created processes. The most notable feature of MKBcs can be seen in its ORIENT rule, which can virtually orient an equation in both directions in parallel in some (often many) processes simulated by the procedure. This mechanism together with the framework of MKB and appropriate heuristics for choosing the most promising processes turns out to be most effective for suppressing the combinatorial explosion. The experimental results show that the performances of MKBcs and SLOTHROP, the existing constraint-based system, are incomparable. Actually, MKBcs has solved some problems more efficiently than SLOTHROP.

The paper is organized as follows. We review the multi-completion in Sect. 2 and the constraint-based completion in Sect. 3. In Sect. 4, we present MKBcs and establish its soundness and completeness. In Sect. 5, we present a variant of MKBcs suitable for using the dependency-pair method for termination checking. In Sect. 6, we discuss the implementation, and in Sect. 7, we report the results of the experiments and discuss the effectiveness of the new procedure. Sect. 8 contains the conclusion and possible future work.

2. Multi-Completion Procedures

Given a set $\mathcal{E}_0$ of equations and a reduction order $\succ$, the standard completion procedure KB tries to compute a convergent set $\mathcal{R}_\omega$ of rewrite rules that is contained in $\succ$ and that induces the same equational theory as $\mathcal{E}_0$.

Starting from the initial state $(\mathcal{E}_0, \mathcal{R}_0)$, where $\mathcal{R}_0 = \emptyset$, the procedure obeys the inference system defined in Fig. 1 to generate a sequence $(\mathcal{E}_0, \mathcal{R}_0) \vdash (\mathcal{E}_1, \mathcal{R}_1) \vdash \cdots$ of deduction[†]. The role of each inference rule is as follows. DELETE removes a trivial equation. ORIENT takes from $\mathcal{E}$ an equation that can be oriented by the reduction order $\succ$, and adds the resultant rule to $\mathcal{R}$. SIMPLIFY reduces an equation using $\mathcal{R}$. COMPOSE reduces the right-hand side of a rule. COLLAPSE reduces the left-hand side of a rule and adds the result as an equation to $\mathcal{E}$, where $\rhd$ in the rule is the *encompassment* order, i.e. $s \rhd l$ iff some subterm of s is an instance of l but not vice versa. DEDUCE adds a critical pair in $\mathcal{R}$ to $\mathcal{E}$ as an equation.

Let $S = (\mathcal{E}_0, \mathcal{R}_0) \vdash (\mathcal{E}_1, \mathcal{R}_1) \vdash \ldots$ be a (finite or infinite) deduction sequence of KB. An equation $s \approx t$ is *persisting* if there exists $i (i \geq 0)$ such that for all $j (j \geq i), s \approx t \in \mathcal{E}_j$. The set of all persisting equations is denoted by $\mathcal{E}_\omega$. Similarly, a rewrite rule $s \to t$ is persisting if there exists $i (i \geq 0)$ such that for all $j (j \geq i), s \to t \in \mathcal{R}_j$. The set of all persisting rewrite rules is denoted by $\mathcal{R}_\omega$. We say that the sequence S is *successful* if $\mathcal{E}_\omega$ is empty and $\mathcal{R}_\omega$ is convergent; S is *failing* if $\mathcal{E}_\omega$ is nonempty; and S is *fair* if every critical pair of $\mathcal{R}_\omega$ belongs to some $\mathcal{E}_j (j \geq 0)$. A completion procedure is *correct* if it generates only successful or failing sequences. It is known that this condition holds iff it generates only fair or failing sequences.

DELETE:	$(\mathcal{E} \cup \{s \approx s\}, \mathcal{R}) \vdash (\mathcal{E}, \mathcal{R})$
ORIENT:	$(\mathcal{E} \cup \{s \approx t\}, \mathcal{R}) \vdash (\mathcal{E}, \mathcal{R} \cup \{s \to t\})$ if $s \succ t$
SIMPLIFY:	$(\mathcal{E} \cup \{s \approx t\}, \mathcal{R}) \vdash (\mathcal{E} \cup \{s \approx u\}, \mathcal{R})$ if $t \to_{\mathcal{R}} u$
COMPOSE:	$(\mathcal{E}, \mathcal{R} \cup \{s \to t\}) \vdash (\mathcal{E}, \mathcal{R} \cup \{s \to u\})$ if $t \to_{\mathcal{R}} u$
COLLAPSE:	$(\mathcal{E}, \mathcal{R} \cup \{s \to t\}) \vdash (\mathcal{E} \cup \{u \approx t\}, \mathcal{R})$ if $l \to r \in \mathcal{R}, s \to_{\{l \to r\}} u$, and $s \rhd l$
DEDUCE:	$(\mathcal{E}, \mathcal{R}) \vdash (\mathcal{E} \cup \{s \approx t\}, \mathcal{R})$ if $u \to_{\mathcal{R}} s$ and $u \to_{\mathcal{R}} t$

Fig. 1 Inference rules of KB.

A *multi-completion* procedure accepts as input a finite set $O = \{\succ_1, \ldots, \succ_m\}$ of reduction orders as well as a set $\mathcal{E}_0$ of equations. The mission of the procedure is basically the same as KB: it tries to compute a convergent set $\mathcal{R}_\omega$ of rewrite rules that is contained in *some* $\succ_i$ and that induces the same equational theory as $\mathcal{E}_0$. To achieve this mission, the multi-completion procedure simulates the execution of m parallel processes $P = \{P_1, \ldots, P_m\}$, with P_i executing KB for the reduction order $\succ_i$ and the common input $\mathcal{E}_0$.

For efficient simulation of multiple processes, the multi-completion procedure MKB developed in [10] exploits the data structure called nodes and represents the state of the procedure by a set of nodes. Let $I = \{1, 2, \ldots, m\}$ be the set of indexes for orders in O (also for processes in P). A *node* is a tuple $\langle s : t, R_0, R_1, E \rangle$, where $s : t$ (called a *datum*) is an ordered pair of terms, and R_0, R_1, and E (called *labels*) are subsets of I satisfying the following condition (called *label condition*):

- $R_0 \cap R_1 = R_1 \cap E = E \cap R_0 = \emptyset$ and
- $i \in R_0$ implies $s \succ_i t$, and $i \in R_1$ implies $t \succ_i s$.

Intuitively, R_0 (R_1) denotes the set of indexes of processes in which the current set of rules contains a rule $s \to t$ ($t \to s$). Similarly, E denotes the set of indexes of processes in which the current set of equations contains an equation $s \approx t$. The node $\langle s : t, R_0, R_1, E \rangle$ is considered to be identical with the node $\langle t : s, R_1, R_0, E \rangle$.

The MKB procedure is defined by the inference system working on a set N of nodes, as given in Fig. 2, where the relation $t \doteq l$ in REWRITE-1 means that t is an instance of l and vice versa: i.e., t and l are syntactically the same up to renaming variables. DELETE, ORIENT, and DEDUCE simulate the corresponding inference rules of KB, respectively. REWRITE-1 simulates the SIMPLIFY and COMPOSE rules of KB. REWRITE-2 simulates the SIMPLIFY, COMPOSE, and COLLAPSE rules of KB. GC and SUBSUME are called optional rules: they do not necessarily simulate KB, but can affect the efficiency of MKB. Starting from the initial set of nodes,

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

[†]In practice, the set union in the left-hand side of the inference rules should be interpreted as the disjoint union, as explicitly denoted by the symbol $\cup'$ in [8]. In a theoretical setting, however, most authors prefer the symbol $\cup$, as there is no problem (at least theoretically) if it is interpreted as the standard union.

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

REWRITE-2: $N \cup \{\langle s : t, R_0, R_1, E\rangle\} \vdash N \cup$
$\left\{ \begin{array}{l} \langle s : t, R_0 \setminus R, R_1 \setminus R, E \setminus R\rangle \\ \langle s : u, R_0 \cap R, \emptyset, (R_1 \cup E) \cap R\rangle \end{array} \right\}$
if $\langle l : r, R, \ldots, \ldots\rangle \in N, t \rightarrow_{\{l\to r\}} u$,
$t \rhd l$, 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 $\langle l : r, R, \ldots, \ldots\rangle \in N$,
$\langle l' : r', R', \ldots, \ldots\rangle \in N, R \cap R' \neq \emptyset$,
$u \rightarrow_{\{l\to r\}} s$, and $u \rightarrow_{\{l'\to r'\}} t$

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
$E'' = (E \setminus (R'_0 \cup R'_1)) \cup (E' \setminus (R_0 \cup R_1))$

Fig. 2 Inference rules of MKB.

the procedure generates a sequence $N_0 \vdash N_1 \vdash \cdots$. In the semantics of MKB, the following definition of *projections* relates the information on nodes to the states of processes.

Definition 2.1: 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

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

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

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

The definitions above are extended for a set N of nodes, as follows:

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

Intuitively, $\mathcal{E}[N, i]$ ($\mathcal{R}[N, i]$) denotes the set of equations (rewrite rules) contained in the simulated process P_i, when the state of MKB is N.

If, for some N and i, $\mathcal{E}[N, i]$ is empty and all critical pairs of $\mathcal{R}[N, i]$ have been created, MKB returns $\mathcal{R}[N, i]$ as the final result, as the semantics of MKB shows that it is the convergent system obtained from the successful KB sequence computed by the process P_i. In this case, we call the index i the *successful index*. The notions of fairness and correctness are discussed in [10]. We illustrate a sample execution of MKB in the following example.

Example 2.2: Let $\mathcal{E}_0 = \{a \approx b, b \approx c\}$ and $O = \{\succ_1, \ldots, \succ_4\}$ where each reduction order is a partial order on $\{a, b, c\}$ induced by the following specification.

$$a \succ_1 b \succ_1 c \qquad a \succ_2 b, c \succ_2 b$$
$$b \succ_3 a \succ_3 c \qquad b \succ_4 a, c \succ_4 a$$

Let $I = \{1, \ldots, 4\}$. The initial set of nodes is

$$N_0 = \{n_1 = \langle a : b, \emptyset, \emptyset, I\rangle, n_2 = \langle b : c, \emptyset, \emptyset, I\rangle\}.$$

By applying ORIENT to n_1 (in both directions), we obtain

$$N_1 = \{n'_1 = \langle a : b, \{1, 2\}, \{3, 4\}, \emptyset\rangle, n_2\}.$$

By applying REWRITE-1 to n_2 using $b \rightarrow a$ (from n'_1) as a rewrite rule, we obtain

$$N_2 = \{n'_1, n'_2 = \langle b : c, \emptyset, \emptyset, \{1, 2\}\rangle, n_3 = \langle a : c, \emptyset, \emptyset, \{3, 4\}\rangle\}.$$

This means that in the processes P_3 and P_4, the equation $b \approx c$ has been reduced to $a \approx c$. By applying ORIENT to n'_2, we obtain

$$N_3 = \{n'_1, n''_2 = \langle b : c, \{1\}, \{2\}, \emptyset\rangle, n_3\}.$$

At this point, the state of the process P_2 is $(\mathcal{E}[N_3, 2], \mathcal{R}[N_3, 2]) = (\emptyset, \{a \rightarrow b, c \rightarrow b\})$. Since $\mathcal{E}[N_3, 2]$ is empty and $\mathcal{R}[N_3, 2]$ is convergent, MKB returns $\mathcal{R}[N_3, 2]$ as the result.

For the convenience of the interested readers, we continue the execution of the procedure. By applying REWRITE-1 to n'_1 using $b \rightarrow c$ (from n''_2) as a rewrite rule, followed by the application of SUBSUME to n_3 and a temporarily-created node $\langle a : c, \{1\}, \emptyset, \emptyset\rangle$, we obtain

$$N_4 = \{n''_1 = \langle a : b, \{2\}, \{3, 4\}, \emptyset\rangle, n''_2, n'_3 = \langle a : c, \{1\}, \emptyset, \{3, 4\}\rangle\}.$$

At this point, P_1 have succeeded in obtaining another convergent system $\{a \rightarrow c, b \rightarrow c\}$. By applying ORIENT to n'_3, we obtain

$$N_5 = \{n''_1, n''_2, n''_3 = \langle a : c, \{1, 3\}, \{4\}, \emptyset\rangle\}.$$

At this point, P_4 have succeeded in obtaining yet another result $\{b \rightarrow a, c \rightarrow a\}$.

3. Completion Procedures with Constraint Systems

Wehrman, Stump, and Westbrook proposed a new completion procedure which requires no reduction orders as input [18]. We call this procedure KBcs. KBcs ensures the termination of the generated systems using a termination checker instead of the given reduction order. In order to check the existence of a reduction order for ensuring the

DELETE:	$(\mathcal{E} \cup \{s \approx s\}, \mathcal{R}, \mathcal{C}) \vdash (\mathcal{E}, \mathcal{R}, \mathcal{C})$
ORIENT:	$(\mathcal{E} \cup \{s \approx t\}, \mathcal{R}, \mathcal{C}) \vdash$ $(\mathcal{E}, \mathcal{R} \cup \{s \to t\}, \mathcal{C} \cup \{s \to t\})$ if $\mathcal{C} \cup \{s \to t\}$ terminates
SIMPLIFY:	$(\mathcal{E} \cup \{s \approx t\}, \mathcal{R}, \mathcal{C}) \vdash$ $(\mathcal{E} \cup \{s \approx u\}, \mathcal{R}, \mathcal{C})$ if $t \to_{\mathcal{R}} u$
COMPOSE:	$(\mathcal{E}, \mathcal{R} \cup \{s \to t\}, \mathcal{C}) \vdash$ $(\mathcal{E}, \mathcal{R} \cup \{s \to u\}, \mathcal{C})$ if $t \to_{\mathcal{R}} u$
COLLAPSE:	$(\mathcal{E}, \mathcal{R} \cup \{s \to t\}, \mathcal{C}) \vdash$ $(\mathcal{E} \cup \{u \approx t\}, \mathcal{R}, \mathcal{C})$ if $l \to r \in \mathcal{R}, s \to_{\{l \to r\}} u$ and $s \rhd l$
DEDUCE:	$(\mathcal{E}, \mathcal{R}, \mathcal{C}) \vdash (\mathcal{E} \cup \{s \approx t\}, \mathcal{R}, \mathcal{C})$ if $u \to_{\mathcal{R}} s$ and $u \to_{\mathcal{R}} t$

Fig. 3 Inference rules of KBcs.

termination, the procedure keeps a *constraint system*, represented by a set of rewrite rules consisting of all previously added rewrite rules, because checking the termination of only the current $\mathcal{R}$ results in an unsound system [14]. Suppose an equation is orientable in both directions. Such a situation never arises in KB, but in KBcs, the system can orient the equation in either direction. This choice is non-deterministic, but in practical implementations, the system should eventually orient the equation in both directions in order to ensure the completeness of the search algorithm.

KBcs is defined by the inference system in Fig. 3 working on a triple $(\mathcal{E}, \mathcal{R}, \mathcal{C})$, where $\mathcal{E}$ is a set of equations, and $\mathcal{R}$ and $\mathcal{C}$ are sets of rewrite rules. We write $(\mathcal{E}, \mathcal{R}, \mathcal{C}) \vdash_{KBcs} (\mathcal{E}', \mathcal{R}', \mathcal{C}')$ if the latter is obtained from the former by one application of an inference rule of KBcs. The constraint system $\mathcal{C}$ accumulates rewrite rules each time an equation is oriented, but unlike $\mathcal{R}$, it never removes any rewrite rules if a rule is removed from $\mathcal{R}$ in COMPOSE or COLLAPSE. If $\mathcal{C}$ is terminating, the transitive closure of the reduction relation $\to_{\mathcal{C}}^{+}$ is a reduction order containing $\mathcal{R}$.

KBcs has two advantages over KB. First, KBcs need not force the users to input any reduction orders. Instead of using reduction orders explicitly, KBcs implicitly constructs the reduction order $\to_{\mathcal{C}}^{+}$ by checking the termination of $\mathcal{C}$. KBcs can benefit from various fully-automated termination checkers for checking the termination. Second, KBcs can exploit modern termination proving methods such as the dependency-pair method. Classical termination proving methods are based on the *local* orientation check for each rewrite rule with the given reduction order. This simplifies the ORIENT inference rule in the classical completion procedures. Some modern termination proving methods, on the other hand, have an ability of *global* termination analysis that considers structural relationship among rewrite rules. This often makes the modern methods more powerful than the classical methods in terms of termination proving abilities. Of course, the modern methods tend to take more time than the classical methods, but this is due to the fact that the modern methods typically call the classical methods (many times); so what really costs time within the modern methods are the classical methods. Note, however, that the ability of global termination analysis can be a disadvantage for completion, because we cannot reuse a termination proof for $\mathcal{R}$ when proving termination of $\mathcal{R} \cup \{s \to t\}$. In order to reduce the difficulty caused by this disadvantage, we will present a special inference system based on the dependency-pair method in Sect. 5 and a technique for caching termination conditions in Sect. 6.2.

For finite execution, KBcs is sound, but for infinite execution, KBcs may be unsound because termination of each intermediate constraint system $\mathcal{C}_i$ does not imply termination of their union $\bigcup_{i \geq 0} \mathcal{C}_i$ [18]. However, KBcs is complete in the sense that if a successful KB sequence exists, KBcs can simulate it.

Since KBcs is unsound for infinite execution, we only consider finite execution. Let $S = (\mathcal{E}_0, \mathcal{R}_0, \mathcal{C}_0) \vdash_{KBcs} (\mathcal{E}_1, \mathcal{R}_1, \mathcal{C}_1) \vdash_{KBcs} \ldots \vdash_{KBcs} (\mathcal{E}_n, \mathcal{R}_n, \mathcal{C}_n)$ be a finite deduction sequence of KBcs. By removing the constraint systems, we can obtain the deduction sequence $S' = (\mathcal{E}_0, \mathcal{R}_0) \vdash (\mathcal{E}_1, \mathcal{R}_1) \vdash \cdots \vdash (\mathcal{E}_n, \mathcal{R}_n)$ of KB. We say that S is *successful* (*failing*, *fair*) if S' is successful (failing, fair). A completion procedure of KBcs is *correct* if it generates only fair or failing sequences of KBcs.

4. Constraint-Based Multi-Completion Procedures

In this section we present a new procedure MKBcs which simulates multiple execution of KBcs in the framework of MKB.

4.1 Basic Idea

As described in the previous section, KB orients an equation in the direction determined by the given reduction order, but KBcs can orient the equation in either direction and this decision is non-deterministic. Thus, in terms of the tree-search algorithms, we have two branches to explore. When the completeness of the search algorithms is required, we can think of at least two types of major implementation schemes for the non-determinism, based on the so-called "don't know" non-determinism and the "don't care" non-determinism. In the "don't know" non-determinism, the procedure is implemented as a single sequential process which explores an arbitrary branch for the moment but can backtrack to this decision point later in order to explore the other branches. SLOTHROP [18] is such an implementation based on the best-first search (rather than the backtracking or the depth-first) strategy in connection with a cost function that evaluates tree nodes in order to determine which branch to explore next.

In the "don't care" non-determinism, on the other hand, the procedure is implemented as a single sequential process which explores an arbitrary branch and never backtracks. If the completeness is required, the overall system should be implemented as a set of concurrent processes which correspond one-to-one with the set of branches. Each process commits itself to exploring the given branch and never backtracks. Combined with a fair scheduler for the execution of processes, the search procedure is ensured to be complete. The semantics of MKBcs we will present in this section is

based on this non-determinism.

In reality, the multi-completion procedure MKBcs never creates multiple concurrent processes but only simulates a part of their concurrent execution efficiently in a single process. To capture the general idea intuitively, consider a process P_1 executing KBcs based on the "don't care" non-determinism. Let $(\mathcal{E} \cup \{s \approx t\}, \mathcal{R}, C)$ be the current state of the process P_1 and suppose that P_1 is about to make a decision on the orientation of the equation $s \approx t$ which can be oriented in both directions. In this situation, we assume that P_1 splits itself into two processes P_2 and P_3. More exactly, after the split operation, P_1 will vanish, P_2 will start from the initial state $(\mathcal{E}, \mathcal{R} \cup \{s \to t\}, C \cup \{s \to t\})$, and P_3 will start from $(\mathcal{E}, \mathcal{R} \cup \{t \to s\}, C \cup \{t \to s\})$. P_1 is called the parent of P_2 and P_3, and P_2 and P_3 are called the children of P_1.

This computation can be simulated by the framework of the multi-completion in a way similar to MKB, except that the interpretation of the notion of "processes" should be different. In the semantics of MKB, a "process" means a computational process which, for a given reduction order, generates a KB deduction sequence. In the semantics of MKBcs, however, it means one which, for a given branch, generates a (part of) KBcs deduction sequence based on the "don't care" non-determinism. Another difference is that in MKB the number of processes is fixed by the number of the given reduction orders, while in MKBcs the number of processes is variable because of the dynamic nature of process creation. This affects the representation of indexes for processes. In MKB, each process is identified with a natural number, called the index. In MKBcs, however, each process is identified with a bit string, say, $b_1 b_2 \dots b_n$. The bit string is also called the index in MKBcs. If the process with the index $b_1 b_2 \dots b_n$ has two children, their indexes are defined to be $b_1 b_2 \dots b_n 0$ and $b_1 b_2 \dots b_n 1$, respectively. With this notation, we can obtain the index of the parent process by removing the rightmost bit of the children.

Let us illustrate how MKBcs can simulate the splitting of processes. In MKBcs, a node structure is extended to a 6-tuple $\langle s : t, R_0, R_1, E, C_0, C_1 \rangle$, where $s : t$ is a pair of terms (as in MKB) and the labels R_0, R_1, E, C_0, and C_1 are sets of indexes (bit strings) of processes. The purpose of the labels is almost the same as that of the labels in MKB except that the interpretation of processes are different (as discussed before) and we have introduced new labels C_0 and C_1 for keeping constraint systems. Suppose the system has a process P_1 (with the index 1) which is to be split into two processes P_{10} and P_{11} (with the indexes 10 and 11, respectively) when orienting an equation $s \approx t$. The splitting is simulated by the following operations on nodes in MKBcs. First, we modify the target node $\langle s : t, R_0, R_1, E \cup \{1\}, C_0, C_1 \rangle$ to $\langle s : t, R_0 \cup \{10\}, R_1 \cup \{11\}, E, C_0 \cup \{10\}, C_1 \cup \{11\} \rangle$. This ensures that the process P_{10} contains the rewrite rule $s \to t$, the process P_{11} contains $t \to s$, and the process P_1 vanishes. Moreover, for each node other than the target node, we replace the index 1 in all the labels by two indexes 10 and 11. For example, the node $\langle a : b, R_0 \cup \{1\}, R_1, E, C_0 \cup \{1\}, C_1 \rangle$ is modified to $\langle a : b, R_0 \cup \{10, 11\}, R_1, E, C_0 \cup \{10, 11\}, C_1 \rangle$. This ensures that the equations and rewrite rules contained in the process P_1 are also contained in the processes P_{10} and P_{11}.

4.2 Bit String Encoding for Process Indexes

Let us formally define the bit string encoding for indexes. Let $\mathcal{P}$ be the set of all bit strings. We will associate each KBcs process with an element of $\mathcal{P}$ in the following way, and in the rest of the papers, identify processes (previously denoted by $P_{b_1 b_2 \dots b_n}$) with their indexes (the bit strings). We assume that KBcs starts with the initial process denoted by the empty bit string ϵ. Let p be (the index of) a KBcs process and suppose p is to be split into two children when orienting an equation. We define the indexes of those children as $p0$ and $p1$ by concatenating p with a bit 0 and 1, respectively. In the process $p0(p1)$, the equation is oriented from left to right (from right to left) and we add the resultant rewrite rule to the current constraint system.

One may imagine a binary tree to intuitively understand the encoding. Each process corresponds to a leaf of the tree, and each leaf is associated with a bit string showing how one can get there from the root by following the bits on outgoing edges one by one at each non-terminal node (go left if the bit is 0 and go right otherwise).

When $p = b_1 b_2 \dots b_n$ is a bit string, the strings $b_1 b_2 \dots b_j (j \in \{0, 1, \dots, n\})$ are called the prefixes of p. In particular, the empty bit string ϵ (for $j = 0$) is a prefix of p. The prefixes other than p are *proper* prefixes.

The *concatenation* of a bit string p and a bit b is denoted by pb. Conversely, we define the *cut* function by $cut(pb) = p$ and $cut(\epsilon) = \epsilon$.

This idea of encoding processes by bit strings is formally described in the notion of *well-encoding* defined as follows.

Definition 4.1 (well-encoding): A set of bit strings Q is *well-encoded* if for every $p \in Q$, Q contains no proper prefixes of p.

For example, $Q = \{0, 10, 11\}$ is a well-encoded set. The following proposition shows two basic properties of well-encodings. The easy proofs are omitted.

Proposition 4.2: Let Q be a well-encoded set. Then:

(1) Every subset of Q is well-encoded.
(2) If $p \in Q$, then $p0 \notin Q$ and $p1 \notin Q$.

The second part of this proposition ensures that we can create a *fresh* bit string (for representing an index for a dynamically-created new process) by the concatenation of $p \in Q$ and a bit. By using this property, we can introduce an operation for *splitting* a process as follows.

Definition 4.3 (splitting): Let Q be a well-encoded set and p be a bit string. Then we define the function $split_p(Q)$ as follows.

$$split_p(Q) = \begin{cases} Q \setminus \{p\} \cup \{p0, p1\}, & \text{if } p \in Q \\ Q & \text{otherwise} \end{cases}$$

Note that the two sets $Q \setminus \{p\}$ and $\{p0, p1\}$ are disjoint, by Proposition 4.2 (2).

Let P be a well-encoded set. Then the definition above is extended to the function $split_P(Q)$ defined as follows.

$$split_P(Q) = Q \setminus P \cup \{p0, p1 \mid p \in P \cap Q\}$$

Note that only the bit strings contained in both P and Q are removed from Q and *split* into two fresh strings. The remaining strings of Q are still contained in $split_P(Q)$. For example, if $P = \{0, 10, 111\}$ and $Q = \{0, 10, 110\}$, then $split_P(Q) = \{00, 01, 100, 101, 110\}$.

In the binary tree interpretation, splitting corresponds to the operation of attaching two children $p0$ and $p1$ to the leaf p of the tree associated with Q (if $p \in Q$). The following lemma ensures that splitting preserves the well-encoding property.

Lemma 4.4: If Q is well-encoded, then $split_p(Q)$ is well-encoded.

Proof. The case $p \notin Q$ is trivial. Consider the case $p \in Q$ and suppose Q is well-encoded. Then Q contains no proper prefixes of p. By the definition, $split_p(Q)$ does not contain p. Therefore, $split_p(Q)$ contains no proper prefixes of $p0$ and $p1$. Thus if $split_p(Q)$ were not well-encoded, $split_p(Q)$ would contain a proper prefix q of some $q' \in Q \setminus \{p\}$. Since $Q \setminus \{p\}$ is well-encoded, q must be either $p0$ or $p1$. However, this implies that p is a proper prefix of $q' \in Q$. This contradicts our assumption that Q is well-encoded. □

This lemma can be easily lifted to the general case as follows.

Lemma 4.5: Let P be a set of bit strings. If Q is well-encoded, then $split_P(Q)$ is well-encoded.

The *ancestor* function defined below is needed for *rewinding* the splitting operation.

Definition 4.6 (ancestor function): Let q be a bit string and P be a set of bit strings. The *direct ancestor* of q with respect to P is defined by

$$anc_P(q) = \begin{cases} cut(q) & \text{if } cut(q) \in P \\ q & \text{otherwise} \end{cases}$$

The following two lemmas are just technical and used in some proofs later (often implicitly).

Lemma 4.7: Let Q be well-encoded and $P \subseteq Q$. Then $q \in Q \Rightarrow anc_P(q) = q$.

Proof. Since Q is well-encoded, if $q \in Q$, then $cut(q) \notin Q$, thus $cut(q) \notin P$. Therefore, $anc_P(q) = q$. □

Lemma 4.8: Let Q be well-encoded and $P \subseteq Q$. Then

(1) $q \in split_P(Q) \Rightarrow anc_P(q) \in Q$ for all bit strings q.
(2) $anc_P(q) \in Q \Rightarrow q \in split_P(Q)$ for all $q \notin P$.

Proof. (1) If $\exists p \in P$ such that $q \in \{p0, p1\}$, then $cut(q) = p \in P$, thus $anc_P(q) = p \in Q$. Otherwise, we have $q \in Q$ and $anc_P(q) = q \in Q$ by Lemma 4.7.

(2) If $\exists p \in P$ such that $q \in \{p0, p1\}$, then $anc_P(q) = p$ and $split_P(Q)$ contains both $p0$ and $p1$, so $q \in split_P(Q)$. Otherwise, we have $anc_P(q) = q$ and $q \in Q$. From the assumption $q \notin P$, we have $q \in split_P(Q)$. □

4.3 MKBcs

In order to develop MKBcs in the framework of MKB, we extend the definition of the node structure by adding two labels C_0, C_1 for keeping constraint systems.

Definition 4.9 (node): A node n in MKBcs is a 6-tuple $\langle s : t, R_0, R_1, E, C_0, C_1 \rangle$, where the labels R_0, R_1, E, C_0, and C_1 are well-encoded sets of bit strings satisfying the following label condition:

- $(R_0 \cup C_0) \cap (R_1 \cup C_1) = \emptyset$
- $E \cap (R_0 \cup R_1 \cup C_0 \cup C_1) = \emptyset$

We denote a node by n and a set of nodes by N, and assume $n = \langle s : t, R_0, R_1, E, C_0, C_1 \rangle$ unless explicitly specified. The node n is considered to be identical with the node $\langle t : s, R_1, R_0, E, C_1, C_0 \rangle$. We denote the set of bit strings (representing processes) $R_0 \cup R_1 \cup E \cup C_0 \cup C_1$ occurring in a node n by $\mathcal{P}(n)$ and define $\mathcal{P}(N) = \bigcup_{n \in N} \mathcal{P}(n)$.

Definition 4.10 ($\mathcal{E}$- and $\mathcal{R}$-projections): Let N be a set of nodes, and p a process (a bit string). The $\mathcal{E}$- and $\mathcal{R}$-projections of N onto p are defined as follows:

$$\mathcal{E}[N, p] = \bigcup_{n \in N} \mathcal{E}[n, p]$$

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

$$\mathcal{R}[N, p] = \bigcup_{n \in N} \mathcal{R}[n, p]$$

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

Definition 4.11 (C-projection and constraints): Let N be a set of nodes, and p a process (a bit string). The C-projection $C[N, p]$ of N onto p is defined as follows:

$$C[N, p] = \bigcup_{n \in N} C[n, p]$$

$$C[n, p] = \begin{cases} \{s \to t\}, & \text{if } p \in C_0, \\ \{t \to s\}, & \text{if } p \in C_1, \\ \emptyset, & \text{otherwise.} \end{cases}$$

A process p *satisfies the constraints* in N if $C[N, p]$ is terminating. A set of nodes N *satisfies the constraints* if for all p in $\mathcal{P}(N)$, $C[N, p]$ is terminating.

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

ORIENT: $N \cup \{n\} \vdash split_P(N) \cup \{n'\}$
if $n = \langle s : t, R_0, R_1, E, C_0, C_1\rangle$,
$n' = \langle s : t, R_0 \cup R_{lr}, R_1 \cup R_{rl}, E',$
$C_0 \cup R_{lr}, C_1 \cup R_{rl}\rangle$,
$E_{lr}, E_{rl} \subseteq E, E_{lr} \cup E_{rl} \neq \emptyset$,
$P = E_{lr} \cap E_{rl}, E' = E \setminus (E_{lr} \cup E_{rl})$,
$C[N, p] \cup \{s \to t\}$ terminates for all $p \in E_{lr}$,
$C[N, p] \cup \{t \to s\}$ terminates for all $p \in E_{rl}$,
$R_{lr} = (E_{lr} \setminus E_{rl}) \cup \{p0 \mid p \in P\}$,
and $R_{rl} = (E_{rl} \setminus E_{lr}) \cup \{p1 \mid p \in P\}$

REWRITE-1: $N \cup \{\langle s : t, R_0, R_1, E, C_0, C_1\rangle\} \vdash$
$$N \cup \left\{ \begin{array}{l} \langle s : t, R_0 \setminus R, R_1, \\ E \setminus R, C_0, C_1\rangle \\ \langle s : u, R_0 \cap R, \emptyset, \\ E \cap R, \emptyset, \emptyset\rangle \end{array} \right\}$$
if $\langle l : r, R, \ldots, \ldots, \ldots, \ldots\rangle \in N$,
$t \to_{\{l \to r\}} u, t \doteq l$,
and $(R_0 \cup E) \cap R \neq \emptyset$

REWRITE-2: $N \cup \{\langle s : t, R_0, R_1, E, C_0, C_1\rangle\} \vdash$
$$N \cup \left\{ \begin{array}{l} \langle s : t, R_0 \setminus R, R_1 \setminus R, \\ E \setminus R, C_0, C_1\rangle \\ \langle s : u, R_0 \cap R, \emptyset, \\ (R_1 \cup E) \cap R, \emptyset, \emptyset\rangle \end{array} \right\}$$
if $\langle l : r, R, \ldots, \ldots, \ldots, \ldots\rangle \in N$,
$t \to_{\{l \to r\}} u, t \rhd l$,
and $(R_0 \cup R_1 \cup E) \cap R \neq \emptyset$

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

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

SUBSUME: $N \cup \left\{ \begin{array}{l} \langle s : t, R_0, R_1, E, C_0, C_1\rangle, \\ \langle s' : t', R'_0, R'_1, E', C'_0, C'_1\rangle \end{array} \right\}$
$\vdash N \cup \{\langle s : t, R_0 \cup R'_0, R_1 \cup R'_1,$
$E'', C_0 \cup C'_0, C_1 \cup C'_1\rangle\}$
if $s : t$ and $s' : t'$ are variants and
$E'' = (E \setminus (R'_0 \cup R'_1 \cup C'_0 \cup C'_1)) \cup$
$(E' \setminus (R_0 \cup R_1 \cup C_0 \cup C_1))$.

Fig. 4 Inference rules of MKBcs.

The C-projection $C[N, p]$ computes the constraint system maintained in the process p. The definition 4.3 of the split function is extended for a node n and a set of nodes N as follows.

$$split_P(n) = \langle s : t, split_P(R_0), split_P(R_1), split_P(E), split_P(C_0), split_P(C_1)\rangle$$

$$split_P(N) = \{split_P(n) \mid n \in N\}$$

Based on this notation, the inference rules of MKBcs are given in Fig. 4. MKBcs works on a set of extended nodes. The general idea is almost the same as MKB, and as in MKB, the GC and SUBSUME rules are optional rules. The key change lies in the ORIENT rule. This rule works as follows. The system focuses attention on a node n and for each process p in the E label of n, it tries to orient the equation $s \approx t$ (stored in n as a datum) while satisfying the constraints. More precisely, if $C[N, p] \cup \{s \to t\}$ is terminating, p is collected in a set E_{lr}. Similarly, if $C[N, p] \cup \{t \to s\}$ is terminating, p is collected in E_{rl}. Then $P = E_{lr} \cap E_{rl}$ denotes the set of processes in which the equation is orientable in both directions. All of such processes p are split into $p0$ and $p1$ for orienting from left to right and vice versa. Finally, a new node n' is created by modifying the labels of n. The processes $E_{lr} \cup E_{rl}$ are removed from the E label, and the processes R_{lr} (R_{rl}) in which the equation is oriented from left to right (from right to left) are added to the R_0 and C_0 (R_1 and C_1) labels.

Let N and N' be two sets of nodes. We write $N \vdash_{MKBcs} N'$ if the latter is obtained from the former by one application of an inference rule of MKBcs. Given a set $\mathcal{E}_0$ of equations, MKBcs starts from the initial set of nodes $N_0 = \{\langle s : t, \emptyset, \emptyset, \{\epsilon\}, \emptyset, \emptyset\rangle \mid s \approx t \in \mathcal{E}_0\}$ since we start with the single (root) process denoted by the empty bit string ϵ. MKBcs generates a sequence $N_0 \vdash_{MKBcs} N_1 \vdash_{MKBcs} \cdots$. Let N be a state of the generation process (i.e., $N = N_i$ for some i). MKBcs keeps the following conditions invariant.

- $\mathcal{P}(N)$ is a well-encoded set.
- Every node of N satisfies the label condition.
- N satisfies the constraints.

The proofs are straightforward by using the following lemmas and induction. (The easy proofs are omitted.)

Lemma 4.12: If $N \vdash_{MKBcs} N'$ and $\mathcal{P}(N)$ is well-encoded, then $\mathcal{P}(N')$ is also well-encoded.

Lemma 4.13: If $N \vdash_{MKBcs} N'$ and every node of N satisfies the label condition, then every node of N' also satisfies the label condition.

Lemma 4.14: If $N \vdash_{MKBcs} N'$ and N satisfies the constraints, then N' also satisfies the constraints.

If a process p has succeeded in obtaining a convergent system at a state N, that is, $\mathcal{E}[N, p]$ is empty and all critical pairs of $\mathcal{R}[N, p]$ have been created, MKBcs can return the convergent system $\mathcal{R}[N, p]$ as the final result. In this case, we call the process (index) p the *successful index*. Let us illustrate a sample execution of MKBcs.

Example 4.15: Let $\mathcal{E}_0 = \{a \approx b, b \approx c\}$. The initial set of nodes is

$$N_0 = \{n_1 = \langle a : b, \emptyset, \emptyset, \{\epsilon\}, \emptyset, \emptyset\rangle, n_2 = \langle b : c, \emptyset, \emptyset, \{\epsilon\}, \emptyset, \emptyset\rangle\}.$$

By applying ORIENT to n_1, we obtain:

$$N_1 = \{n'_1 = \langle a : b, \{0\}, \{1\}, \emptyset, \{0\}, \{1\}\rangle, n'_2 = \langle b : c, \emptyset, \emptyset, \{0, 1\}, \emptyset, \emptyset\rangle\}$$

By applying REWRITE-1 to n'_2 using $b \to a$ (from n'_1) as a rewrite rule, we obtain:

$$N_2 = \{n'_1, n''_2 = \langle b : c, \emptyset, \emptyset, \{0\}, \emptyset, \emptyset\rangle, n_3 = \langle a : c, \emptyset, \emptyset, \{1\}, \emptyset, \emptyset\rangle\}$$

By applying ORIENT to n_2'', we obtain:

$$N_3 = \{n_1'' = \langle a : b, \{00, 01\}, \{1\}, \emptyset, \{00, 01\}, \{1\}\rangle, \\ n_2''' = \langle b : c, \{00\}, \{01\}, \emptyset, \{00\}, \{01\}\rangle, n_3\}$$

At this point, we see that $\mathcal{E}[N_3, 01] = \emptyset$ and no critical pairs can be created from $\mathcal{R}[N_3, 01] = \{a \to b, c \to b\}$. Therefore, P_{01} have succeeded in obtaining the convergent system $\mathcal{R}[N_3, 01]$.

We only present a brief sketch of the proof of the soundness of the ORIENT rule. The soundness means that the ORIENT rule of MKBcs correctly simulates the ORIENT rule of KBcs in some processes q and has no effect on other processes. This situation is described in the following lemma by introducing the symbol $\vdash^{=}_{KBcs}$ for denoting the reflexive closure of $\vdash_{KBcs}$.

Lemma 4.16 (Soundness of ORIENT):
If $N \vdash_{MKBcs} N'$ by the ORIENT rule, then there exists $P \subseteq \mathcal{P}(N)$ such that for all $q' \in \mathcal{P}(N')$,

$$(\mathcal{E}[N, q], \mathcal{R}[N, q], C[N, q]) \vdash^{=}_{KBcs} \\ (\mathcal{E}[N', q'], \mathcal{R}[N', q'], C[N', q'])$$

where $q = anc_P(q')$.

Proof. In this proof, we use the symbol N_i for referring to N in the ORIENT rule. Taking E_{lr}, E_{rl}, P, R_{lr}, R_{rl} and E' as specified in the ORIENT rule, we let $N = N_i \cup \{n\}, N' = split_P(N_i) \cup \{n'\}$, $n = \langle s : t, R_0, R_1, E, C_0, C_1\rangle$ and $n' = \langle s : t, R_0 \cup R_{lr}, R_1 \cup R_{rl}, E', C_0 \cup R_{lr}, C_1 \cup R_{rl}\rangle$. By the definition of $\mathcal{E}$-projection,

$$\mathcal{E}[N, q] = \mathcal{E}[N_i, q] \cup \mathcal{E}[n, q]$$

and

$$\mathcal{E}[N', q'] = \mathcal{E}[split_P(N_i), q'] \cup \mathcal{E}[n', q'].$$

Since equations other than $s \approx t$ (stored as a datum of n) are untouched, they are preserved in all processes, thus formally we have

$$\mathcal{E}[N_i, q] = \mathcal{E}[split_P(N_i), q'].$$

(By the way, this is true if N_i contains another node with datum $s : t$.) We denote this set by $\mathcal{E}$. Likewise, $\mathcal{R}[N_i, q] = \mathcal{R}[split_P(N_i), q']$ and $C[N_i, q] = C[split_P(N_i), q']$ and we denote them by $\mathcal{R}$ and C, respectively. Then the inference we should verify for this lemma is written as

$$(\mathcal{E} \cup \mathcal{E}[n, q], \mathcal{R} \cup \mathcal{R}[n, q], C \cup C[n, q]) \vdash^{=}_{KBcs} \\ (\mathcal{E} \cup \mathcal{E}[n', q'], \mathcal{R} \cup \mathcal{R}[n', q'], C \cup C[n', q']).$$

We consider three cases:

(Case 1) We assume $q' \in R_{lr}$. This implies that $\mathcal{R}[n', q'] = C[n', q'] = \{s \to t\}$ and $\mathcal{E}[n', q'] = \emptyset$. We will show that $q = anc_P(q') \in E$. By $q' \in R_{lr}$, either $q' \in E_{lr} \setminus E_{rl}$ or $q' \in \{p0, p1 \mid p \in P\}$ must hold. If $q' \in E_{lr} \setminus E_{rl}$ then $anc_P(q') = q'$, thus we have $q = q'$ and $q \in E_{lr} \subseteq E$. On the other hand, if $q' = pb$ for some $p \in P$ and a bit b, then $q = anc_P(q') = p \in P \subseteq E$. Therefore, in either case, we have $q \in E$, and thus $\mathcal{E}[n, q] = \{s \approx t\}, \mathcal{R}[n, q] = C[n, q] = \emptyset$. It follows that MKBcs has simulated the KBcs inference

$$(\mathcal{E} \cup \{s \approx t\}, \mathcal{R}, C) \vdash_{KBcs} \\ (\mathcal{E}, \mathcal{R} \cup \{s \to t\}, C \cup \{s \to t\}),$$

and $C \cup \{s \to t\}$ is terminating since $q' \in R_{lr}$.

(Case 2) We assume $q' \in R_{rl}$. In this case, we can follow the arguments similar to Case 1 to verify that $(\mathcal{E} \cup \{s \approx t\}, \mathcal{R}, C) \vdash_{KBcs} (\mathcal{E}, \mathcal{R} \cup \{t \to s\}, C \cup \{t \to s\})$ and $C \cup \{t \to s\}$ is terminating.

(Case 3) We assume $q' \notin (R_{lr} \cup R_{rl})$. This is combined with $q' \notin P$ to get $q' \notin (E_{lr} \cup E_{rl})$ and $q' \notin \{p0, p1 \mid p \in P\}$. From the last condition, we see that $anc_P(q') = q'$ and thus $q = q'$. Therefore, $\mathcal{E}[n, q] = \mathcal{E}[n', q']$, $\mathcal{R}[n, q] = \mathcal{R}[n', q']$ and $C[n, q] = C[n', q']$. It follows that $(\mathcal{E}[N, q], \mathcal{R}[N, q], C[N, q]) = (\mathcal{E}[N', q'], \mathcal{R}[N', q'], C[N', q'])$. □

We have implicitly used the ancestor function $anc_P(q)$ for relating the processes before and after the application of the ORIENT rule so far. Note, however, that it may be also associated with other rules without the splitting operation, because by setting $P = \emptyset$, we have $anc_P(q) = q$. Actually, before and after the application of those rules, the indexes of the processes should be unchanged. The following two theorems exploit this extension to make the descriptions concise.

Theorem 4.17 (Soundness of MKBcs):
If $N \vdash_{MKBcs} N'$, then there exists a set $P \subseteq \mathcal{P}(N)$ such that for all $p' \in \mathcal{P}(N')$,

$$(\mathcal{E}[N, p], \mathcal{R}[N, p], C[N, p]) \vdash^{=}_{KBcs} \\ (\mathcal{E}[N', p'], \mathcal{R}[N', p'], C[N', p'])$$

where $p = anc_P(p')$. The strict part, $\vdash_{KBcs}$, holds for at least one p' if the employed rule is not optional.

Theorem 4.18 (Completeness of MKBcs):
If $(\mathcal{E}[N, p], \mathcal{R}[N, p], C[N, p]) \vdash_{KBcs} (\mathcal{E}', \mathcal{R}', C')$, then there exists a set N' of nodes, $P \subseteq \mathcal{P}(N)$, and $p' \in \mathcal{P}(N')$ such that $p = anc_P(p')$, $\mathcal{E}' = \mathcal{E}[N', p'], \mathcal{R}' = \mathcal{R}[N', p'], C' = C[N', p']$, and $N \vdash_{MKBcs} N'$.

Finally, we discuss the fairness of MKBcs. Since KBcs is unsound for infinite execution, we only consider finite execution of MKBcs. Let $S = N_0 \vdash_{MKBcs} N_1 \vdash_{MKBcs} \cdots \vdash_{MKBcs} N_n$ be a finite deduction sequence of MKBcs. By using the soundness and the projections, we can obtain a sequence $S' = (\mathcal{E}_0, \mathcal{R}_0, C_0) \vdash^{=}_{KBcs} (\mathcal{E}_1, \mathcal{R}_1, C_1) \vdash^{=}_{KBcs} \cdots \vdash^{=}_{KBcs} (\mathcal{E}_n, \mathcal{R}_n, C_n)$, where $\mathcal{E}_j = \mathcal{E}[N_j, p_j], \mathcal{R}_j = \mathcal{R}[N_j, p_j], C_j = C[N_j, p_j] (j \in \{0, 1, \ldots, n\})$, and p_j are defined by

$$p_j = anc_{P_j}(p_{j+1})$$

for some $P_j \subseteq \mathcal{P}(N_j) (j \in \{0, 1, \ldots, n-1\})$ and some $p_n \in$

$\mathcal{P}(N_n)$. By removing all the equivalent steps $(\mathcal{E}_i, \mathcal{R}_i, C_i) = (\mathcal{E}_{i+1}, \mathcal{R}_{i+1}, C_{i+1})$, we get a proper deduction sequence S'' of KBcs (as described in [10]). Let KBcs[S] be the set of all such deduction sequences of KBcs induced from S. We define the success, failure, and fairness of S as follows:

- S is *successful* if there exists a successful deduction sequence in KBcs[S]. (In this case, the process $p_n \in \mathcal{P}(N_n)$ is called the successful index.)
- S is *failing* if all deduction sequences of KBcs[S] are failing.
- S is *fair* if there exists a non-failing, fair deduction sequence in KBcs[S].

A procedure for generating deduction sequences of MKBcs is *correct* if it generates only fair or failing sequences. Suppose we have a non-failing, fair deduction sequence S of MKBcs. Then there exists in KBcs[S] a non-failing, fair deduction sequence of KBcs. If the sequence ends with an element $(\mathcal{E}_\omega, \mathcal{R}_\omega, C_\omega)$ with $\mathcal{E}_\omega = \emptyset$, then $\mathcal{R}_\omega$ is the convergent system, the final result of MKBcs.

5. Constraint-Based Multi-Completion Procedures with the Dependency-Pair Method

In this section, we present a more efficient variant (referred to as MKBdp) of MKBcs when we use the *dependency-pair* method [1], [7], [12] for termination checking. The emphasis is on how we can take advantage of additional node structures to improve the efficiency of MKBcs.

We begin by reviewing some basic notions on the dependency-pair method. Let $\mathcal{R}$ be a set of rewrite rules over a set of function symbols Σ, and let $\Sigma^\# = \Sigma \cup \{f^\# \mid f \in \Sigma\}$. If $s = f(s_1, \ldots, s_n)$, then $s^\# = f^\#(s_1, \ldots, s_n)$. We denote the root symbol of a term s by $root(s)$. The set of all defined symbols of $\mathcal{R}$ is defined by $D(\mathcal{R}) = \{root(l) \mid l \to r \in \mathcal{R}\}$. Let $Sub(t)$ be the set of all subterms of t and $PSub(t)$ be the set of all proper subterms of t. We define $Sub_s(t) = Sub(t) \setminus PSub(s)$ and $\mathcal{SP}(\mathcal{R}) = \{s \to u \mid s \to t \in \mathcal{R}, u \in Sub_s(t)\}$. The elements of $SP(\mathcal{R})$ are called *subterm-pairs*. Then the set of *dependency-pairs* of $\mathcal{R}$ is defined by $\mathcal{DP}(\mathcal{R}) = \{s^\# \to u^\# \mid s \to u \in \mathcal{SP}(\mathcal{R}), root(s) \in D(\mathcal{R}), root(u) \in D(\mathcal{R})\}$.

Now, we introduce new node structures which will turn out to be helpful for handling dependency pairs.

Definition 5.1 (subterm-pair node): A *subterm-pair node* is a pair $\langle s \to u, Q\rangle$ of a rewrite rule $s \to u$ and a set of bit strings Q (representing processes).

Definition 5.2 (defined-symbol node):
A *defined-symbol node* is a pair $\langle f, Q\rangle$ of a function symbol f and a set of bit strings Q (representing processes).

Intuitively, the subterm-pair node claims that the rewrite rule $s \to u$ is contained in $\mathcal{SP}(\mathcal{R})$ of all processes of Q, and the defined-symbol node claims that f is a defined symbol in all processes of Q.

Let us define the inference rules of MKBdp working on a tuple $\langle N, SP, D\rangle$, where N is a set of (extended) nodes, SP is a set of subterm-pair nodes, and D is a set of defined-symbol nodes. When MKBcs derives N' from N, MKBdp derives $\langle N', SP', D'\rangle$ from $\langle N, SP, D\rangle$. In almost all cases, we define $SP' = SP$ and $D' = D$. The exceptional case is when the Orient rule has been applied in MKBcs. Suppose that an equation $s \approx t$ has been oriented from left to right to create a rewrite rule $s \to t$ in a process p in R_{lr}. Then for each subterm $u \in Sub_s(t)$ of t, we have a subterm-pair $s \to u$ to be added in $\mathcal{SP}(\mathcal{R})$. By considering all such processes of R_{lr}, this situation can be represented by the set of subterm-pair nodes $\{\langle s \to u, R_{lr}\rangle \mid u \in Sub_s(t)\}$. Similar consideration is needed for the case in which a rewrite rule $t \to s$ is created in processes p in R_{rl}. In addition, we have to split processes P (in which the equation has been oriented in both directions, as defined in the Orient rule) stored in the labels Q of the existing subterm-pair nodes. Combining all of these operations, we get

$$\begin{aligned} SP' = \{\langle l \to r, split_P(Q)\rangle \mid \langle l \to r, Q\rangle \in SP\} \\ \cup \{\langle s \to u, R_{lr}\rangle \mid u \in Sub_s(t)\} \\ \cup \{\langle t \to u, R_{rl}\rangle \mid u \in Sub_t(s)\} \end{aligned}$$

To see how we can maintain the defined-symbol nodes, suppose again that $s \to t$ has been created in a process p in R_{lr}. Then the symbol $root(s)$ must be a defined symbol in the process p. Therefore, we have to look up a defined-symbol node $\langle f, Q\rangle$ in D such that $f = root(s)$ and add p in the label Q. By considering all such processes of R_{lr}, the set of processes to be added in the label Q of $\langle f, Q\rangle$ is R_{lr} if $f = root(s)$, and $\emptyset$ otherwise. Similarly, we have to consider the case of the opposite orientation $(t \to s)$. In addition, we have to consider the splitting. Considering all of these operations, we get

$$\begin{aligned} D' = \{\langle f, split_P(Q) \cup D^f(s, R_{lr}) \cup D^f(t, R_{rl})\rangle \\ \mid \langle f, Q\rangle \in D\} \end{aligned}$$

where

$$D^f(s, Q) = \begin{cases} Q & \text{if } f = root(s) \\ \emptyset & \text{otherwise} \end{cases}$$

and other symbols s, t, R_{lr}, R_{rl}, and P denote those symbols defined in the Orient rule.

MKBdp starts from the initial tuple $\langle N_0, SP_0, D_0\rangle$ where N_0 is the initial set of nodes of MKBcs, $SP_0 = \emptyset$, and $D_0 = \{\langle f, \emptyset\rangle \mid f \in \Sigma\}$. We write $\langle N, SP, D\rangle \vdash_{MKBdp} \langle N', SP', D'\rangle$ if the latter is obtained from the former by one application of an inference rule of MKBdp. Let $S = \langle N_0, SP_0, D_0\rangle \vdash_{MKBdp} \langle N_1, SP_1, D_1\rangle \vdash_{MKBdp} \cdots \vdash_{MKBdp} \langle N_n, SP_n, D_n\rangle$ be a finite deduction sequence of MKBdp. Let MKBcs[S] $= N_0 \vdash_{MKBcs} N_1 \vdash_{MKBcs} \cdots \vdash_{MKBcs} N_n$ be the deduction sequence of MKBcs obtained by taking the first elements N_i from each tuple of S. We say that S is *successful* (*fair*, *failing*) if MKBcs[S] is successful (fair, failing). A procedure for generating deduction sequences of MKBdp is *correct* if it generates only fair or failing sequences. Suppose we have a non-failing, fair deduction sequence S of

MKBdp. Then there exists in KBcs[MKBcs[S]] a non-failing, fair deduction sequence of KBcs. If the sequence ends with an element $(\mathcal{E}_\omega, \mathcal{R}_\omega, C_\omega)$ with $\mathcal{E}_\omega = \emptyset$, then $\mathcal{R}_\omega$ is the convergent system, the final result of MKBdp.

We define a projection for relating the state of MKBdp to the set of dependency-pairs maintained in a process. The projection $\mathcal{DP}[SP, D, p]$ is defined as follows. For each subterm-pair node $\langle s \to u, P\rangle$ in SP such that $p \in P$, we check to see if the root symbols of s and u are the defined symbols in the process p. This can be checked by looking up the defined-symbol nodes $\langle root(s), Q\rangle$ and $\langle root(u), Q'\rangle$ in D. A dependency-pair $s^\# \to u^\#$ is collected in $\mathcal{DP}[SP, D, p]$ if and only if $p \in Q$ and $p \in Q'$. Thus this procedure is summarized concisely as follows:

$$\begin{aligned}\mathcal{DP}[SP, D, p] = \{s^\# \to u^\# \mid \langle s \to u, P\rangle \in SP,\\ \langle root(s), Q\rangle \in D, \langle root(u), Q'\rangle \in D,\\ p \in P \cap Q \cap Q'\}\end{aligned}$$

Theorem 5.3:
Let $\langle N_0, SP_0, D_0\rangle \vdash_{MKBdp} \langle N_1, SP_1, D_1\rangle \vdash_{MKBdp} \cdots$ be a deduction sequence of MKBdp. For every $i \geq 0$ and $q \in \mathcal{P}(N_i)$, $\mathcal{DP}(C[N_i, q]) = \mathcal{DP}[SP_i, D_i, q]$.

This theorem ensures that we can obtain all dependency-pairs of all processes by maintaining SP and D, instead of calculating $\mathcal{DP}(C[N, p])$ from scratch. The easy proof is omitted.

Example 5.4: Let $\langle N, SP, D\rangle$ be the MKBdp state defined as follows:

$$\begin{aligned}N = \{&n_1 = \langle f(x) : g(h(x)), \{0\}, \{1\}, \emptyset, \{0\}, \{1\}\rangle,\\ &n_2 = \langle g(x) : h(x), \emptyset, \emptyset, \{0, 1\}, \emptyset, \emptyset\rangle\}\\ SP = \{&\langle f(x) \to g(h(x)), \{0\}\rangle,\\ &\langle f(x) \to h(x), \{0\}\rangle,\\ &\langle g(h(x)) \to f(x), \{1\}\rangle\}\\ D = \{&\langle f, \{0\}\rangle, \langle g, \{1\}\rangle, \langle h, \emptyset\rangle\}\end{aligned}$$

By orienting the node n_2, we obtain the state $\langle N', SP', D'\rangle$ as follows:

$$\begin{aligned}N' = \{&n'_1 = \langle f(x) : g(h(x)),\\ &\{00, 01\}, \{10, 11\}, \emptyset, \{00, 01\}, \{10, 11\}\rangle,\\ &n'_2 = \langle g(x) : h(x),\\ &\{00, 10\}, \{01, 11\}, \emptyset, \{00, 10\}, \{01, 11\}\rangle\}\\ SP' = \{&\langle f(x) \to g(h(x)), \{00, 01\}\rangle,\\ &\langle f(x) \to h(x), \{00, 01\}\rangle,\\ &\langle g(h(x)) \to f(x), \{10, 11\}\rangle,\\ &\langle g(x) \to h(x), \{00, 10\}\rangle,\\ &\langle h(x) \to g(x), \{01, 11\}\rangle\}\\ D' = \{&\langle f, \{00, 01\}\rangle, \langle g, \{00, 10, 11\}\rangle, \langle h, \{01, 11\}\rangle\}\end{aligned}$$

6. Implementation

In this section, we briefly describe our implementation of MKBcs. The techniques of caching conditions for reduction orders are also described.

```
procedure mkbcs(E) {
    N_o := {⟨s : t, ∅, ∅, {ε}, ∅, ∅⟩ | s ≈ t ∈ E}
    N_c := ∅
    while success(N_o, N_c) = false {
        if N_o = ∅ { return(fail) }
        n := choose(N_o, N_c)
        N_o := N_o \ {n}
        if n ≠ ⟨..., ∅, ∅, ∅, ..., ...⟩ {
            while orient(n, N_o, N_c) = true {
                N_o := union(N_o, rewrite(N_c, {n}))
                N_o := union(N_o, deduce(n, N_c))
                N'_o := rewrite(N_o, union(N_c, {n}))
                N_o := union(N_o, N'_o)
            }
            N_c := union(N_c, {n})
        }
    }
    return (R[N_c, p]) where p = success(N_o, N_c)
}
```

Fig. 5 Implementation of MKBcs.

6.1 Pseudo Code for MKBcs

A possible MKBcs completion procedure is given in Fig. 5 as an imperative pseudo code. The set N of all nodes, which represents the state of MKBcs, is partitioned into two sets of nodes: the *open* set N_o and the *closed* set N_c as in [10]. Let us refer to DELETE, ORIENT, and GC as the *single-node* operations, and REWRITE(-1, -2), DEDUCE, and SUBSUME as the *double-node* operations. The former is applied to a single node, while the latter to a pair of nodes. Initially, N_o contains all nodes and N_c is empty. The outer **while** loop of Fig. 5 keeps the following conditions invariant.

- Every node in N_c has been fully considered for application of the single-node operations.
- Every pair of nodes in N_c has been fully considered for application of the double-node operations.

Thus the major mission of the loop is to take a node n from N_o and apply the single-node operations to it, followed by the application of the double-node operations with n supplied as an argument. Finally, n is moved to N_c in order to preserve the loop invariant.

The $success(N_o, N_c)$ function checks if there exists a successful process, i.e., a process p such that p is not contained in any labels of N_o nodes and any E labels of N_c nodes. If such p exists, the function returns p; otherwise, it returns false. Note that for such p, $\mathcal{R}[N_c, p]$ is the convergent set to be returned from MKBcs.

The $choose(N_o, N_c)$ function selects a node from N_o. The strategy for node selection heavily affects the number of inference steps required for leading processes to success. Based on the idea of [18], we define a cost function

$$cost(\mathcal{E}, \mathcal{R}, C) = |\mathcal{E}| + |CP(\mathcal{R})| + |C|$$

for evaluating the states $(\mathcal{E}, \mathcal{R}, C)$ of KBcs processes, where

$CP(\mathcal{R})$ is the set of all critical pairs of $\mathcal{R}$. In MKBcs, this function is used to define the cost function for evaluating the states of processes $p \in \mathcal{P}(N)$ as follows.

$$cost(p) = cost(\mathcal{E}[N, p], \mathcal{R}[N, p], C[N, p])$$

where $N = N_o \cup N_c$. (Note that *choose* takes N_c as the second argument in order to compute the cost function above, unlike the *choose* of [10] with only one argument N_o.) We prefer a process p with smaller cost containing unoriented equations, because processes with smaller cost would have less possibility of divergence, and the orientation is necessary for executing the inner **while** loop which contains the essential inference steps of MKBcs in its body.

Based on this idea, we have developed a cost function for evaluating nodes as follows.

$$cost(n) = \begin{cases} (min_{p \in E} cost(p), |s| + |t|) & \text{if } E \neq \emptyset \\ (\infty, |s| + |t|) & \text{if } E = \emptyset \end{cases}$$

where $n = \langle s : t, \ldots, \ldots, E, \ldots, \ldots \rangle$, and $|s|$ ($|t|$) is the number of occurrences of function symbols and variables in s (t), respectively. The symbol ∞ denotes a natural number which is large enough to ensure that the nodes with non-empty E labels are preferred.

Note that the cost of a node n is represented by a pair of natural numbers. The pairs should be compared in the lexicographic order of elements (from left to right): when the first elements are the same, the ties are broken with the second elements.

The procedure $rewrite(N, N')$ repeatedly applies Rewrite(-1,-2) rules 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 those inferences. The labels of the nodes of N are directly modified in order to have the updated nodes stored in the same memory location. For example, when $N = \{\langle a : b, \{0, 1\}, \emptyset, \emptyset, C_0, C_1 \rangle\}$ and $N' = \{\langle b : c, \{1\}, \emptyset, \emptyset, C'_0, C'_1 \rangle\}$, the procedure *rewrite* modifies N to $\{\langle a : b, \{0\}, \emptyset, \emptyset, C_0, C_1 \rangle\}$ and returns the result $\{\langle a : c, \{1\}, \emptyset, \emptyset, C_0, C_1 \rangle\}$. The $union(N, N')$ function computes the union of N and N' and applies the Subsume rule as much as possible such that at least one node is taken from N'. The $deduce(n, N)$ function applies the Deduce rule to $\{n\} \cup N$ and returns the set of all critical nodes (i.e., the nodes which capture the critical pairs in some processes) between n and a node from $\{n\} \cup N$.

The specifications of the procedures described above are almost the same as those of MKB in [10]. The main difference lies in the procedure $orient(n, N_o, N_c)$, which accepts a node n (containing a datum $s : t$) together with N_o and N_c, and tries to orient the equation $s \approx t$ for each process in the E label of n. N_o and N_c are used to extract the constraints for those processes. In MKB, the direction of the equation in each process is uniquely determined by the reduction order stored in that process. In MKBcs, however, we have to consider the potential existence of processes in which the equation can be oriented in both directions. Our implementation of Orient is given in Fig. 6,

```
procedure orient(n, N_o, N_c) {
    assume n = ⟨s : t, R_0, R_1, E, C_0, C_1⟩
    E_lr, E_rl := ∅
    for each p ∈ E {
        if C[N_o ∪ N_c, p] ∪ {s → t} terminates E_lr := E_lr ∪ {p}
        if C[N_o ∪ N_c, p] ∪ {t → s} terminates E_rl := E_rl ∪ {p}
        /* if E_lr ∪ E_rl ≠ ∅ break */
    }
    E := E \ (E_lr ∪ E_rl)
    P := E_lr ∩ E_rl
    R_lr := (E_lr \ E_rl) ∪ {q0 | q ∈ P}
    R_rl := (E_rl \ E_lr) ∪ {q1 | q ∈ P}
    R_0 := R_0 ∪ R_lr
    R_1 := R_1 ∪ R_rl
    C_0 := C_0 ∪ R_lr
    C_1 := C_1 ∪ R_rl
    N_o := split_P(N_o)
    N_c := split_P(N_c)
    if E_lr ∪ E_rl ≠ ∅ return true else return false
}
```

Fig. 6 Implementation of *orient*.

where we assume all arguments of this procedure are mutable (i.e. can be modified by the side effect of the assignment statements). Note that the potential processes mentioned above are collected in a set P and split into $q0$ and $q1$ for each q in P. The processes in which the equation can be oriented only from left to right (from right to left) are collected in R_{lr}(R_{rl}). The labels of the node are updated by appropriate assignment. Also, N_o and N_c are updated by the splitting. Finally, the procedure returns true if and only if the equation has been oriented (in either way) in at least one process. For example, when $n = \langle a : b, \emptyset, \emptyset, \{\epsilon\}, \emptyset, \emptyset \rangle$, $N_o = \{\langle b : c, \emptyset, \emptyset, \{\epsilon\}, \emptyset, \emptyset \rangle\}$, and $N_c = \emptyset$, the procedure *orient* modifies n to $\{\langle a : b, \{0\}, \{1\}, \emptyset, \{0\}, \{1\} \rangle\}$, N_o to $\{\langle b : c, \emptyset, \emptyset, \{0, 1\}, \emptyset, \emptyset \rangle\}$, and returns the result **true**. N_c remains to be the empty set.

The line enclosed by /* and */ should be skipped as a comment in order to continue the loop for computing the maximal E_{lr} and E_{rl} that satisfy the conditions specified in the Orient rule. By collecting in $E_{lr} \cup E_{rl}$ all processes that can orient the equation $s \approx t$, we can exploit the pseudo-parallelism and the node structure of the MKBcs system most effectively in later stages, as all those processes can share the rewrite rule $s \to t$ or $t \to s$. In the next section, however, this line will be uncommented and executed just for experimental reasons. In this case, the loop will terminate as soon as $E_{lr} \cup E_{rl}$ becomes a singleton set, so there will be only a single process that orients the equation $s \approx t$.

6.2 Caching Conditions for Reduction Orders

In general, for proving a proposition p in the given logical system L, automated theorem provers often transform p into a proposition in another logical system M. Let us denote this proposition by $[\![p]\!]$. The two propositions are logically equivalent in the sense that p is true in L iff $[\![p]\!]$ is true in M. Computationally, however, it is often expected that proving $[\![p]\!]$ in M should take less CPU time than proving p in L.

We can use this idea in the proof of termination as well. For example, a proposition $f(a) \succ_{LPO} b$ in the theory of the lexicographic path orders (LPO) may be transformed to $[\![f(a) \succ_{LPO} b]\!] = X_{ab} \vee X_{fb}$ in the classical, propositional logic as in [11]. As another example, when we consider the polynomial orders, the resultant proposition might be a Diophantine constraint $x_0^f + x_1^f x_0^a > x_0^b$, as discussed in [5].

In the implementation of MKBdp, we use this technique for efficiently finding a reduction order compatible with the dependency pairs. Actually, we cache each proposition $[\![s \succeq t]\!]$ by associating it with the node $\langle s : t, \dots \rangle$ and the subterm-pair node $\langle s : t, P \rangle$. The reason the caching technique is promising is that the constraint system grows incrementally and the propositions once generated for termination of C can be reused later for termination of another system containing C. In addition, the cache can be shared among the processes. Therefore, the caching technique is effective for difficult problems, which require long deduction sequences and many processes.

7. Experimental Results

We have experimented with our implementation of MKBcs on a set of the standard benchmark problems [15] and some difficult problems experimented in [17], [18]. We have not used problems in the *Termination Problem Data Base*† because almost all problems contained in it are already convergent or too easy to complete. For comparative experiments, we have used all examples that we succeeded in completion within 24 hours. However, we have omitted the results of too easy examples (for which the computation time was less than 0.1 seconds). Our built-in termination checker is based on the dependency-pair method. Moreover, in order to find reduction orders for ensuring termination, we have used the combination of polynomial interpretation and SAT solving proposed in [5]. The built-in termination checker has the following features:

- the dependency-graph analysis (finding strongly connected components)
- no use of the subterm criteria
- the argument filtering and usable rules with polynomial orderings [7]
- the linear polynomial orderings with coefficients in $\{0, 1\}$

We refer to our implementation with this termination checker as MKBcs/POL. All experiments have been performed on a workstation equipped with Intel Xeon 2.13 GHz CPU and 1 GB system memory.

7.1 Comparison with Unshared Orientation

Since the most essential, novel part of MKBcs is its Orient rule, we have examined its effectiveness by some experiments. As described in Sect. 6.1, we can maximally share the execution of the Orient rule in several processes by skipping the comment line (as intended). To see the effectiveness of this design, we have compared it with the design in which the execution of the Orient rule is unshared. The latter case, which we call the unshared orientation, can be examined by removing /* and */ from Fig. 6, thus executing the comment line in order to break the loop as soon as the procedure *orient* finds a single process which can orient the equation $s \approx t$ in either way.

Table 1 Comparison with the unshared orientation.

Problem	unshared		shared	
	all	re/de	all	re/de
SK90_3.01	1.0	0.7	0.6	0.3
SK90_3.03	0.6	0.5	0.3	0.2
SK90_3.04	190.3	150.1	55.9	30.2
SK90_3.05	1.7	1.3	0.8	0.5
SK90_3.06	3.6	2.1	2.0	0.6
SK90_3.07	4.1	2.3	2.3	0.6
SK90_3.09	146.6	115.1	29.4	7.2
SK90_3.27	21.1	3.2	19.1	1.7
SK90_3.28	410.5	133.4	207.9	2.1
SK90_3.29	1.0	0.4	0.5	0.0
$WSW07\_GE_1$	1.4	0.7	0.8	0.2
$WSW07\_CGE_2$	435.9	272.4	126.4	10.7
$WSW07\_CGE_3$	-	-	32867.6	568.8
WS06_PR	28074.7	14690.5	10752.1	25.7

The results are summarized in Table 1, where the "unshared" columns show the results for the unshared orientation, and the "shared" columns for the shared orientation (as intended by MKBcs). The "all" columns show the total time (in seconds) and the "re/de" columns show the time consumed by the Rewrite-1,2 and Deduce rules. Hyphens indicate that we could not get the results within 24 hours. We can see that in the shared orientation the total time is shorter (as shown in the "all" columns) and the node-based (shared) rewriting and deducing by Rewrite-1,2 and Deduce are more effective (as shown in the "re/de" columns) especially for the problems requiring a long computation time.

7.2 Comparison of MKBcs/AProVE with Slothrop

We have compared the performance of MKBcs with Slothrop, the first constraint-based completion tool described in [18]. The current implementation of Slothrop uses AProVE [6], one of the most powerful termination checkers known in the literature. In order to compare MKBcs with Slothrop in the same environment, we have developed MKBcs/AProVE, which is MKBcs using AProVE instead of our built-in termination checker, and compared it with Slothrop. AProVE employs various modern termination checking methods, including the dependency-pair method. Thus MKBcs/AProVE is more powerful than MKBcs/POL in terms of the ability to complete equational theories, i.e. if MKBcs/POL succeeds in completing a theory, MKBcs/AProVE also succeeds in completing the same theory.

The results are summarized in Table 2, where the "all" columns show the total time and the "tc" columns show the

†The data base is available from: `http://www.lri.fr/~marche/tpdb/`

Table 2 Comparison of MKBcs with SLOTHROP.

Problem	MKBcs/AProVE		SLOTHROP	
	all	tc	all	tc
SK90_3.01	2.9	89	20.6	326
SK90_3.03	2.3	59	3.3	86
SK90_3.04	464.8	931	2275.1	1466
SK90_3.05	25.5	103	347.4	577
SK90_3.06	63.4	246	993.8	898
SK90_3.07	43.7	218	2722.5	1811
SK90_3.12	1.6	21	3.5	24
SK90_3.18	2.7	35	2.6	24
SK90_3.19	1.7	45	1.6	21
SK90_3.20	3.7	99	2.4	33
SK90_3.21	1.6	35	50.8	141
SK90_3.23	4.2	63	2.5	35
SK90_3.27	428.2	213	253.8	90
SK90_3.28	12962.8	10757	374.4	807
SK90_3.29	10.8	330	2.4	80
WSW07_GE$_1$	3.9	113	5.8	105
WSW07_CGE$_2$	10488.0	12984	457.6	1381

Table 3 Evaluation of MKBdp and caching.

Problem	no soup	no cache	cache	procs
SK90_3.01	0.61	0.59	0.55	13
SK90_3.03	0.31	0.31	0.30	5
SK90_3.04	63.90	60.67	55.71	14
SK90_3.05	0.89	0.86	0.81	8
SK90_3.06	2.14	2.16	1.97	21
SK90_3.07	2.51	2.43	2.26	21
SK90_3.12	0.12	0.11	0.11	3
SK90_3.18	0.20	0.17	0.14	11
SK90_3.19	0.10	0.10	0.08	20
SK90_3.20	0.13	0.12	0.11	32
SK90_3.23	0.19	0.16	0.15	17
SK90_3.27	20.85	20.09	19.08	9
SK90_3.28	353.51	253.25	207.94	791
SK90_3.29	0.61	0.54	0.50	166
WSW07_GE$_1$	0.98	0.93	0.82	15
WSW07_CGE$_2$	199.13	157.73	126.36	167
WSW07_CGE$_3$	53885.42	41256.02	32867.56	2862
WS06_PR	16041.13	13210.79	10752.13	4872

number of calls to the termination checker. The results show that the two systems are incomparable in their performance: for some problems, MKBcs is faster, but for other problems, SLOTHROP is faster. This is because they are based on totally different ideas. SLOTHROP works in a best-first manner in the search space. When an equation can be oriented in both directions, SLOTHROP chooses one of them, based on some heuristics, and basically sticks to that decision until that choice turns out to be less promising than other choices. On the other hand, MKBcs works in a breadth-first manner. When an equation can be oriented in both directions, MKBcs splits processes and tries both directions in parallel. Since this pseudo-parallelism requires some overhead for managing nodes, SLOTHROP is more efficient when its heuristics are appropriate. However, such heuristics are often difficult to design. When the heuristics are inappropriate, there is a chance for MKBcs to be more efficient.

The major reason for the potential efficiency of MKBcs is that the information (equations and rewrite rules) created in one process can be shared by some (often many) other processes in an inexpensive way based on the label computation for the nodes. Thus the computation time to create the same information in those processes can be saved.

In addition, we can think of another reason for the potential efficiency, noting that even if the heuristics are appropriately designed, they work poor when they are provided with poor information. Actually, the shared information can help the heuristic procedures work more effectively when they try to change their focus on a promising process to other, more promising processes, because, thanks to the sharing, the processes would contain richer information which could help them decide the seemingly "best" processes leading to the earliest success.

Apart from the performance, the convergent term rewriting systems generated by the two completion tools are sometimes different from each other, because of the difference in their process selection. These observations mean that both systems have a role to play in efficient completion with automated, modern termination checking.

7.3 Evaluation of MKBdp and Caching

We show the results when we have considered (1) the node-based calculation of dependency-pairs described as MKBdp in Sect. 5 and (2) the cache-based condition checking described in Sect. 6.2. The total CPU time (in seconds) is shown in Table 3, where the "no soup" column shows the results when no node-based techniques have been applied, the "no cache" column shows the results when node-based calculation has been applied with no conditions cached, and the "cache" column shows the results when all conditions have been cached during the node-based calculation. The "procs" column shows the number of all processes. From the results, we can see that the node-based techniques of MKBdp and the caching are effective for improving the performance of MKBcs/POL, especially for the problems that require a long CPU time and a large number of processes such as WSW07_CGE$_3$ and WS06_PR.

A more practical implementation of MKBcs is outlined in our short, system description paper [13]. There is some overlap between the current paper and [13], such as a basic description of MKBcs and a part of experiments, but in [13], the authors introduced the idea of MKBcs only informally and briefly (without proofs and examples) and, instead, described the implementation extensively. The implementation, called mkbTT, is implemented in OCaml and accepts any termination prover that adheres to the format of the International Competitions of Termination Tools [6]. The experimental results, when coupled with TTT [13], are also shown.

In contrast, the current paper has established the theoretical background of MKBcs by developing the formal framework of bit string encoding of processes, the formal semantics based on the "don't care" non-deterministic KBcs processes, and the formal justifications for the soundness, completeness, and correctness of the procedures. Moreover, this paper has presented MKBdp, developed by more tightly

coupling MKBcs with the dependency-pair-based termination provers, as anticipated in [13] as a future work.

8. Conclusion and Future work

We have presented a new multi-completion procedure MKBcs which efficiently simulates parallel execution of constraint-based procedures. The novel techniques involved are (1) the development of well-encoded bit string systems for encoding and maintaining dynamic processes and (2) the new ORIENT rule defined on the extended definition of the node structure. The idea has been further extended for (3) incorporating the dependency-pair method as the associated termination checker. The experiments show that the process sharing scheme of the ORIENT rule is clearly more efficient than the unshared orientation. Superiority of MKBcs/AProVE (our implementation of MKBcs) to SLOTHROP (the well-known implementation of KBcs) depends on the problems. In general, SLOTHROP is more efficient when its heuristics for process selection in the orientation are correct. However, this is not always the case. When the heuristics are inappropriate, MKBcs plays its role in node-based efficient completion with automated, modern termination checking.

As future work, we are planning to incorporate the ideas of AC-completion and unfailing completion into the framework of MKBcs.

Acknowledgments

This work was partially supported by JSPS Grant-in-Aid for Scientific Research (C), No. 19500020.

References

[1] T. Arts and J. Giesl, "Termination of term rewriting using dependency pairs," Theor. Comput. Sci., vol.236, pp.133–178, 2000.
[2] F. Baader and T. Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.
[3] L. Bachmair, Canonical Equational Proofs, Birkhäuser, 1991.
[4] N. Dershowitz and J.-P. Jouannaud, "Rewrite systems," in Handbook of Theoretical Computer Science, ed. J. van Leeuwen, vol.B, pp.243–320, MIT Press, 1990.
[5] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl, "SAT solving for termination analysis with polynomial interpretations," Proc. 10th International Conference on Theory and Applications of Satisfiability Testing (SAT 2007), vol.4501 of Lecture Notes in Computer Science, pp.340–354, 2007.
[6] J. Giesl, P. Schneider-Kamp, and R. Thiemann, "AProVE 1.2: Automatic termination proofs in the dependency pair framework," Proc. 3rd International Joint Conference on Automated Reasoning, vol.4130 of Lecture Notes in Artificial Intelligence, pp.281–286, 2006.
[7] J. Giesl, R. Thiemann, P. Schneider-Kamp, and S. Falke, "Mechanizing and improving dependency pairs," J. Automated Reasoning, vol.37, no.3, pp.155–203, 2006.
[8] J.W. Klop, "Term rewriting systems," in Handbook of Logic in Computer Science, ed. S. Abramsky, D. Gabby, and T. Maibaum, pp.1–116, Oxford University Press, 1992.
[9] D.E. Knuth and P.B. Bendix, "Simple word problems in universal algebras," in Computational Problems in Abstract Algebra, ed. J. Leech, pp.263–297, Pergamon Press, 1970.
[10] M. Kurihara and H. Kondo, "Completion for multiple reduction orderings," J. Automated Reasoning, vol.23, no.1, pp.25–42, 1999.
[11] M. Kurihara and H. Kondo, "Efficient BDD encodings for partial order constraints with application to expert systems in software verification," vol.3029 of Lecture Notes in Computer Science, pp.827–837, 2004.
[12] N. Hirokawa and A. Middeldorp, "Tyrolean termination tool: Techniques and features," Inf. Comput., vol.205, no.4, pp.474–511, 2007.
[13] H. Sato, S. Winkler, M. Kurihara, and A. Middeldorp, "Multi-completion with termination tools (system description)," Proc. 4th International Joint Conference on Automated Reasoning, vol.5195 of Lecture Notes in Computer Science, pp.306–312, 2008.
[14] A. Sattler-Klein, "About changing the ordering during Knuth-Bendix completion," Proc. 11th Annual Symposium on Theoretical Aspects of Computer Science, vol.775 of Lecture Notes in Computer Science, pp.175–186, 1994.
[15] J. Steinbach and U. Kühler, "Check your ordering - termination proofs and problems," Technical Report SR-90-25, Universität Kaiserslautern, 1990.
[16] Terese, Term Rewriting Systems, Cambridge University Press, 2003.
[17] I. Wehrman and A. Stump, "Mining propositional simplification proofs for small validating clauses," Electronic Notes in Theoretical Computer Science, vol.144, no.2, pp.79–91, 2006.
[18] I. Wehrman, A. Stump, and E. Westbrook, "SLOTHROP: Knuth-Bendix completion with a modern termination checker," Proc. 17th International Conference on Rewriting Techniques and Applications, vol.4098 of Lecture Notes in Computer Science, pp.287–296, 2006.

Haruhiko Sato received a BS in information engineering, and a MS in computer science from Hokkaido University in 2005. He is currently a Ph.D. student at the Graduate School of Information Science and Technology, Hokkaido University. His research interests include term rewriting systems, automated theorem proving, and software engineering. He is a member of IPSJ, JSSST.

Masahito Kurihara received a BS in electrical engineering, and a MS and a Ph.D. in information engineering from Hokkaido University, in 1978, 1980 and 1986 respectively. He is currently a professor of information science at Hokkaido University. His research interests include mathematical logic and automated reasoning in computer science and artificial intelligence.

Sarah Winkler received the BSc. and MSc. degrees in Computer Science from the University of Innsbruck, Austria, in 2006 and 2008, respectively. She is currently a PhD student at the Computational Logic Group at the University of Innsbruck, supervised by Prof. Dr. Aart Middeldorp. Continuing in the field of her master studies, her research interests focus on term rewriting with an emphasis on the usage of automatic termination tools in Knuth-Bendix completion.

Aart Middeldorp received the master and doctorate degrees from Vrije Universiteit Amsterdam in 1986 and 1990, respectively. After working for more than 10 years in Japan as visiting research scientist at Hitachi Advanced Research Laboratory and assistant and associate professor at the University of Tsukuba, in 2003 he was appointed full professor of computer science at the University of Innsbruck. His research interests include computational logic and automated reasoning. He is a member of ACM and EATCS.