PAPER

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 a various way. 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 checker. 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 new procedure is clearly more efficient than the naive parallelization. The existing constraint-based system SLOTHROP, which basically employs the depth-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*

1. Introduction

Term rewriting systems [2], [12] 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 [7] and has been improved in a various way [3]. Given a set of rewrite rules $\mathcal{R}$ (or equations $\mathcal{E}$) and a reduction order on a set of terms, the procedure tries to generate a convergent term rewriting system which is equationally equivalent to $\mathcal{R}$ (or $\mathcal{E}$) 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 choise 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 indefinite, divergent computation and inhibit the exploration of the remaining possibilities.

Kurihara and Kondo [8] 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 simgle operation. We call this type of procedure a *multi-completion* procedure.

As another approach to this problem, Wehrman and Stump [14] 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-

Manuscript received March 24, 2008.

[†]The author is with the Graduate School of Information Science and Technology, Hokkaido University

[††]The author is with the Institute of Computer Science, University of Innsbruck

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

based procedures should orient the equations in both directions (in a breadth-first-like manner) 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 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 new procedure is clearly more efficient than the naive parallelization. The existing constraint-based system SLOTHROP, which basically employs the depth-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.

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

2. Multi-Completion Procedures

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

Starting from the initial state $(E_0, R_0) = (\mathcal{E}, \emptyset)$, the procedure obeys the inference system defined in Fig. 1 to generate a sequence $(E_0, R_0) \vdash (E_1, R_1) \vdash \ldots$ of deduction, where $\rhd$ in the COLLAPSE rule is an *encompassment* order. The sequence is said to *succeed* when the set E_ω of persisting equations $\bigcup_{i\geq 0}\bigcap_{j\geq i} E_j$ is empty and the set R_ω of persisting rewrite rules $\bigcup_{i\geq 0}\bigcap_{j\geq i} R_j$ is convergent. The sequence *fails* iff $E_\omega \neq \emptyset$. A completion procedure is *correct* iff every sequence that does not fail succeeds. A sequence generated by a completion procedure is *fair* iff $CP(R_\omega) \subseteq \bigcup_{i\geq 0} E_i$, where $CP(R)$ denotes the set of all critical pairs generated from every pair of rules of R.

A *multi-completion* procedure accepts as input a finite set $O = \{\succ_1, \ldots, \succ_m\}$ of reduction orders as well as a set $\mathcal{E}$ of equations. The mission of the procedure is basically the same as KB: it tries to compute a convergent set R of rewrite rules that is contained in *some* $\succ_i$ and that induces the same equational theory as $\mathcal{E}$.

The multi-completion procedure MKB developed in [8] exploits the data structure called nodes. Let $I = \{1, 2, \ldots, m\}$ be the set of indexes for orders in O. A

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

Fig. 1 Inference rules of KB

node is a tuple $\langle s : t, R_1, R_2, E\rangle$, where $s : t$ (called a *datum*) is an ordered pair of terms, and R_1, R_2 and E (called *labels*) are subsets of I satisfying the following condition (called *label condition*):

- $R_1 \cap R_2 = R_2 \cap E = E \cap R_1 = \emptyset$ and
- $i \in R_1$ implies $s \succ_i t$, and $i \in R_2$ implies $t \succ_i s$.

In [8], labels are denoted by L_1, L_2 and L_3, but in this paper, we slightly abuse the notation when there is no confusion.

The MKB procedure is defined by the inference system working on a set N of nodes, as given in Fig.2. Among those rules, GC and SUBSUME are called optional rules: they do not necessarily simulate KB, but can affect the efficiency of the procedure. Starting from the initial set of nodes,

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

the procedure generates a sequence $N_0 \vdash N_1 \vdash \cdots$. The procedure simulates the execution of the parallel processes $\{P_1, \ldots, P_m\}$, with P_i executing KB for the reduction order $\succ_i$ and the common input $\mathcal{E}$. In the semantics of MKB, the following definition of *projections* relates the information on nodes to the states of processes.

Definition 2.1: Let $n = \langle s : t, R_1, R_2, E\rangle$ be a node and $i \in I$ be an index. The *E-projection* $E[n, i]$ of n onto i is a (singleton or empty) set of equations defined by

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

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

$$R[n, i] = \begin{cases} \{s \to t\}, & \text{if } i \in R_1, \\ \{t \to s\}, & \text{if } i \in R_2, \\ \emptyset, & \text{otherwise.} \end{cases}$$

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

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

From a successful sequence, the convergent set of rewrite rules can be extracted by projecting N_∞ onto a successful index i.

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

ORIENT: $N \cup \{\langle s : t, R_1, R_2, E \cup E'\rangle\} \vdash N \cup \{\langle s : t, R_1 \cup E', R_2, E\rangle\}$ if $E' \neq \emptyset, E \cap E' = \emptyset$, and $s \succ_i t$ for all $i \in E'$

REWRITE-1: $N \cup \{\langle s : t, R_1, R_2, E\rangle\} \vdash N \cup \left\{ \begin{array}{l} \langle s : t, R_1 \setminus R, R_2, E \setminus R\rangle \\ \langle s : u, R_1 \cap R, \emptyset, E \cap R\rangle \end{array} \right\}$ if $\langle l : r, R, \ldots, \ldots\rangle \in N, t \to_{\{l\to r\}} u$, $t \doteq l$, and $(R_1 \cup E) \cap R \neq \emptyset$

REWRITE-2: $N \cup \{\langle s : t, R_1, R_2, E\rangle\} \vdash N \cup \left\{ \begin{array}{l} \langle s : t, R_1 \setminus R, R_2 \setminus R, E \setminus R\rangle \\ \langle s : u, R_1 \cap R, \emptyset, (R_2 \cup E) \cap R\rangle \end{array} \right\}$ if $\langle l : r, R, \ldots, \ldots\rangle \in N, t \to_{\{l\to r\}} u$, $t \rhd l$, and $(R_1 \cup R_2 \cup E) \cap R \neq \emptyset$

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

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

SUBSUME: $N \cup \left\{ \begin{array}{l} \langle s : t, R_1, R_2, E\rangle, \\ \langle s' : t', R'_1, R'_2, E'\rangle \end{array} \right\} \vdash N \cup \{\langle s : t, R_1 \cup R'_1, R_2 \cup R'_2, E''\rangle\}$ if $s : t$ and $s' : t'$ are variants and $E'' = (E \setminus (R'_1 \cup R'_2)) \cup (E' \setminus (R_1 \cup R_2))$

Fig. 2 Inference rules of MKB

3. Completion Procedures with Constraint Systems

Wehrman and Stump proposed a new completion procedure which requires no reduction orders for input [14]. 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 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 R results in an unsound system [10]. 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 nondeterministic, 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 (E, R, C), where E is a set of equations, and R and C are sets of rewrite rules. We write $(E, R, C) \vdash_{KBcs} (E', R', C')$ if the latter is obtained from the former by one application of an inference rule of KBcs. The constraint system C accumulates rewrite rules each time an equation is *oriented*, but unlike R, it never removes any rewrite rules if a rule is removed from R in COMPOSE or COLLAPSE. If C is terminating, the transitive closure of the reduction relation $\to_C^+$ is a reduction order compatible with 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 $\to_C^+$ by checking the termination of 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 modern methods more powerful than classical methods in terms of termination proving abilities. In terms of the computation cost, however, modern methods tend to take more time than classical methods.

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

DELETE: $(E \cup \{s \approx s\}, R, C) \vdash (E, R, C)$

ORIENT: $(E \cup \{s \approx t\}, R, C) \vdash (E, R \cup \{s \to t\}, C \cup \{s \to t\})$ if $C \cup \{s \to t\}$ terminates

SIMPLIFY: $(E \cup \{s \approx t\}, R, C) \vdash (E \cup \{s \approx u\}, R, C)$ if $t \to_R u$

COMPOSE: $(E, R \cup \{s \to t\}, C) \vdash (E, R \cup \{s \to u\}, C)$ if $t \to_R u$

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

DEDUCE: $(E, R, C) \vdash (E \cup \{s \approx t\}, R, C)$ if $u \to_R s$ and $u \to_R t$

Fig. 3 Inference rules 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 Bit string representation for processes

In order to define MKBcs in a formal setting, we first introduce a new representation of processes. Let p be a process in the semantic domain and suppose p is about to make a decision on the orientation of an equation $s \approx t$ which can be oriented in both directions. In this situation, we *split* p into two processes $p0$ and $p1$. In process $p0$ ($p1$), the equation is oriented from left to right (from right to left) and add the resultant rewrite rule to the current set of the constraints. Similar situations may be arising in other processes dealing with $s \approx t$. The new procedure maintains all such processes and inferences in an effective way in a new node structure. As a process may be identified with a sequence of choices on the directions for the orientation, we encode the processes as bit strings $b_1b_2...b_n$ ($n \geq 0$), where b_j is 0 (1) if the j-th choice has oriented the equation from left to right (from right to left). 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 reach 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). We denote the set of all bit strings by $\mathcal{P}$.

When $p = b_1b_2 \ldots b_n$ is a bit string, the strings $b_1b_2 \ldots b_j (j = 0, 1, \ldots, n)$ 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 terms 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-encoding. The easy proofs are omitted.

Proposition 1: 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 clause of this proposition ensures that we can create a *fresh* bit string (for representing an identifier 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 4.2 (splitting): Let Q be a well-encoded set and p be a bit string. Then we define the function $split_p(Q)$ as follows.

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

Note that two sets $Q \setminus \{p\}$ and $\{p0, p1\}$ are disjoint, by Proposition 1 (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 1: 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 2: 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 4.3 (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 3: 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: 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 3.

(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 $q \in split_P(Q)$. □

4.2 MKBcs

In order to develop MKBcs from MKB, we extend the node structure by adding two labels C_1, C_2 for keeping constraint systems. A node n in MKBcs is a 6-tuple $\langle s : t, R_1, R_2, E, C_1, C_2\rangle$, where the labels R_1, R_2, E, C_1 and C_2 are well-encoded sets of bit strings satisfying the following label condition:

- $(R_1 \cup C_1) \cap (R_2 \cup C_2) = \emptyset$
- $E \cap (R_1 \cup R_2 \cup C_1 \cup C_2) = \emptyset$

We denote a node by n and a set of nodes by N, and assume $n = \langle s : t, R_1, R_2, E, C_1, C_2\rangle$ unless explicitly specified. The node n is considered to be identical with the node $\langle t : s, R_2, R_1, E, C_2, C_1\rangle$. We denote the set of bit strings (representing processes) $R_1 \cup R_2 \cup E \cup C_1 \cup C_2$ occurring in a node n by $\mathcal{P}(n)$ and define $\mathcal{P}(N) = \bigcup_{n\in N} \mathcal{P}(n)$.

The C-projection $C[N, p]$ of the set of nodes N onto process p computes the constraint system maintained in process p.

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

$$C[n, p] = \begin{cases} \{s \to t\}, & \text{if } p \in C_1, \\ \{t \to s\}, & \text{if } p \in C_2, \\ \emptyset, & \text{otherwise.} \end{cases}$$

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

The definition of the split function is extended for a node n and a set of nodes N as follows.

$$split_P(n) = \langle s : t, split_P(R_1), split_P(R_2), split_P(E), split_P(C_1), split_P(C_2)\rangle$$

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

Based on this notation, the inference rules of MKBcs are given in Fig. 4. (As usual, the set union in the left-hand side of the inference rules should be interpreted as the disjoint union in practice.) MKBcs works on a set of extended nodes, keeping constraint systems in the fourth and fifth labels in an obvious way. The general idea is almost the same as MKB. However, the key change lies in the ORIENT rule. This rule works as follows. The system focuses 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_1 and C_1 (R_2 and C_2) labels.

Given a set $\mathcal{E}$ of equations, MKBcs starts from the initial set of nodes $N_0 = \{\langle s : t, \emptyset, \emptyset, \{\epsilon\}, \emptyset, \emptyset \mid s \approx t \in \mathcal{E}\}$ and 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 5: If $N \vdash_{MKBcs} N'$ and $\mathcal{P}(N)$ is well-encoded, then $\mathcal{P}(N')$ is also well-encoded.

Lemma 6: 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 7: If $N \vdash_{MKBcs} N'$ and N satisfies the constraints, then N' also satisfies the constraints.

We can verify the correctness and completeness for MKBcs by relating MKBcs to KBcs. The proofs are almost straightforward, and we only present a brief sketch of the proof of the soundness of 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 in other processes. This situation is described in the following lemma by introducing the symbol $\vdash^{=}_{MKBcs}$ for denoting the reflexive closure of $\vdash_{MKBcs}$.

Lemma 8 (Soundness of ORIENT rule):
If $N \vdash_{MKBcs} N'$ by ORIENT rule, then there exists $P \in \mathcal{P}(N)$ such that

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 $n = \langle s : t, R_1, R_2, E, C_1, C_2\rangle$,
$n' = \langle s : t, R_1 \cup R_{lr}, R_2 \cup R_{rl}$,
$E', C_1 \cup R_{lr}, C_2 \cup R_{rl}\rangle$,
$E_{lr}, E_{rl} \subseteq E, E_{lr} \cup E_{rl} \neq \emptyset$,
$P = E_{lr} \cap E_{rl}, E' = E \setminus (E_{lr} \cup E_{rl})$,
$C[N,p] \cup \{s \to t\}$ 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: $N \cup \{\langle s : t, R_1, R_2, E, C_1, C_2\rangle\} \vdash N \cup \left\{ \begin{array}{l} \langle s : t, R_1 \setminus R, R_2, \\ E \setminus R, C_1, C_2\rangle \\ \langle s : u, R_1 \cap R, \emptyset, \\ E \cap R, \emptyset, \emptyset\rangle \end{array} \right\}$
if $\langle l : r, R, \ldots, \ldots, \ldots, \ldots\rangle \in N$,
$t \to_{\{l \to r\}} u, t \doteq l$,
and $(R_1 \cup E) \cap R \neq \emptyset$

REWRITE-2: $N \cup \{\langle s : t, R_1, R_2, E, C_1, C_2\rangle\} \vdash N \cup \left\{ \begin{array}{l} \langle s : t, R_1 \setminus R, R_2 \setminus R, \\ E \setminus R, C_1, C_2\rangle \\ \langle s : u, R_1 \cap R, \emptyset, \\ (R_2 \cup E) \cap R, \emptyset, \emptyset\rangle \end{array} \right\}$
if $\langle l : r, R, \ldots, \ldots, \ldots, \ldots\rangle \in N$,
$t \to_{\{l \to r\}} u, t \rhd l$,
and $(R_1 \cup R_2 \cup E) \cap R \neq \emptyset$

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: $N \cup \left\{ \begin{array}{l} \langle s : t, R_1, R_2, E, C_1, C_2\rangle, \\ \langle s' : t', R'_1, R'_2, E', C'_1, C'_2\rangle \end{array} \right\}$
$\vdash N \cup \{\langle s : t, R_1 \cup R'_1, R_2 \cup R'_2$,
$E'', C_1 \cup C'_1, C_2 \cup C'_2\rangle\}$
if $s : t$ and $s' : t'$ are variants and
$E'' = (E \setminus (R'_1 \cup R'_2 \cup C'_1 \cup C'_2)) \cup$
$(E' \setminus (R_1 \cup R_2 \cup C_1 \cup C_2))$.

Fig. 4 Inference rules of MKBcs

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

for all $q' \in \mathcal{P}(N')$, 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 $N = N_i \cup \{n\}, N' = split_P(N_i) \cup \{n'\}$, $n = \langle s : t, R_1, R_2, E, C_1, C_2\rangle$ and $n' = \langle s : t, R_1 \cup R_{lr}, R_2 \cup R_{rl}, E', C_1 \cup R_{lr}, C_2 \cup R_{rl}\rangle$. By the definition of E-projection,

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

and

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

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

$$E[N_i,q] = E[split_P(N_i),q'].$$

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

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

We consider three cases:

(Case 1) We assume $q' \in R_{lr}$. This implies that $R[n',q'] = C[n',q'] = \{s \to t\}$ and $E[n',q'] = \emptyset$. 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 $E[n,q] = \{s \approx t\}$, $R[n,q] = C[n,q] = \emptyset$. It follows that MKBcs has simulated the KBcs inference

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

and $\mathcal{C} \cup \{s \to t\}$ 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 $(\mathcal{E} \cup \{s \approx t\}, \mathcal{R}, \mathcal{C}) \vdash_{KBcs} (\mathcal{E}, \mathcal{R} \cup \{t \to s\}, \mathcal{C} \cup \{t \to s\})$ and $\mathcal{C} \cup \{t \to s\}$ 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, $E[n,q] = E[n',q']$, $R[n,q] = R[n',q']$ and $C[n,q] = C[n',q']$. It follows that $(E[N,q], R[N,q], C[N,q]) = (E[N',q'], R[N',q'], C[N',q'])$. □

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 identifiers of the processes should be unchanged. The following two theorems exploit this extension to make the descriptions concise.

Theorem 1 (Soundness of MKBcs):
If $N \vdash_{MKBcs} N'$, then there exists a set $P \subseteq \mathcal{P}(N)$ such that

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

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

Theorem 2 (Completeness of MKBcs):
If $(E[N,p], R[N,p], C[N,p]) \vdash_{KBcs} (E', R', C')$, then there exists a set N' of nodes, $P \subseteq \mathcal{P}(N)$ and $p' \in \mathcal{P}(N')$ such that $p = anc_P(p')$, $E' = E[N',p']$, $R' = R[N',p']$, $C' = C[N',p']$ and $N \vdash_{MKBcs} N'$.

Finally, we note the fairness of MKBcs. The infinite execution of MKBcs may be unsound because KBcs may be unsound. Thus we consider only finite sequences for MKBcs. For finite sequences, the fairness of MKBcs can be defined in the same way as MKB.

5. Constraint-Based Multi-Completion Procedures with 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], [9] for termination checking. The emphasis is on how we can take advantage of yet another node structure to improve the efficiency of MKBcs.

We begin by reviewing some basic notions on the dependency-pair method. Let R be a set of rewrite rules over the function symbol 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 R is defined by $D(R) = \{root(l) \mid l \to r \in R\}$. 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 $SP(R) = \{s \to u \mid s \to t \in R, u \in Sub_s(t)\}$. Then the set of *dependency-pair* of R is defined by $DP(R) = \{s^{\#} \to u^{\#} \mid s \to u \in SP(R), root(s) \in D(R), root(u) \in D(R)\}$.

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 subset Q of $\mathcal{P}$.

Definition 5.2 (defined-symbol node):
A *defined-symbol node* is a pair $\langle f, Q\rangle$ of a function symbol f and a subset Q of $\mathcal{P}$.

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

We define MKBdp inference rules working on the tuple $\langle N, SP, D\rangle$, where N is a set of 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 $\langle N, SP, D\rangle$ and in almost all cases we have $SP' = SP$ and $D' = D$. The exceptional case is when the ORIENT rule has been applied in MKBcs. In that case, we have:

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

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

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 define a new projection for relating the state of MKBdp to the set of dependency-pairs maintained in a process. The projection $DP[SP, D, p]$ is defined as follows:

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

Theorem 3:
Let $\langle N_0, SP_0, D_0\rangle \vdash_{MKBdp} \langle N_1, SP_1, D_1\rangle \vdash_{MKBdp} \ldots$ be a sequence of MKBdp inferences. For every $i \geq 0$ and $q \in \mathcal{P}(N_i)$, $DP(C[N_i, q]) = DP[SP_i, D_i, q]$.

This theorem ensures that we can obtain all dependency-pairs of all processes by maintaining SP and D, instead of calculating $DP(C[N,p])$ from scratch.

6. Implementation

In this section, we briefly describe our implementation of MKBcs and caching techniques for efficient termination checking with the dependency-pair method.

6.1 Pseude code for MKBcs

A possible MKBcs completion procedure as an imperative program is given in Fig. 5. There is not much difference from the implementation of MKB in [8]. The main difference is the *orient* function, which accepts a node $n = \langle s : t, R_1, R_2, E, C_1, C_2\rangle$ together with a set of nodes N (partitioned into the *open* set N_o and the *closed* set N_c) and tries to orient the equation $s \approx t$ for each process p contained in E.

Our implementation of *orient* is given in Fig. 6,

where we assume all arguments of *orient* are mutable. The line marked by **(*)** should be skipped in order to share inferences among the processes as much as possible. The line is executed only when we simulate a naive execution of parallel processes, where each process is executed independently.

```
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_o := N_o \ {n}
        if n ≠ ⟨..., ∅, ∅, ∅, ..., ...⟩ {
            while orient(n, N_o, N_c) = true {
                N_o := N_o ∪ rewrite(N_c, {n})
                N_o := N_o ∪ deduce(n, N_c)
                N'_o := rewrite(N_o, N_c ∪ {n})
                N_o := N_o ∪ N'_o
            }
            N_c := N_c ∪ {n}
        }
    }
    return (R[N_c, p]) where p = success(N_o, N_c)
}
```

Fig. 5 Implementation of MKBcs

```
procedure orient(n, N_o, N_c) {
    assume n = ⟨s : t, R_1, R_2, E, C_1, C_2⟩
    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_1 := R_1 ∪ R_lr
    R_2 := R_2 ∪ R_rl
    C_1 := C_1 ∪ R_lr
    C_2 := C_2 ∪ 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* function

6.2 Caching conditions for reduction orders

In order to efficiently find a reduction order compatible with dependency pairs, it is effective to associate an appropriate condition $[\![s \succeq t]\!]$ equivalent to $s \succeq t$ with the node $\langle s : t, \dots \rangle$ and the subterm-pair node $\langle s : t, P \rangle$. Such a condition might be more suitable for automated proof of termination. For example, when we consider the lexicographic path orders (LPO) as a class of reduction orders, such a condition might be a Boolean formula for representing strict orders on function symbols. As another example, when we consider polynomial orders, the condition might be a diophantine constraint generated by interpreting terms. The reason the caching technique is promising is that the constraint system grows incrementally and the condition once generated for termination of C can be reused later for termination of another system containing C. In addition, the cached condition 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 [11] and some difficult problems experimented in [13], [14]. Our built-in termination checker is based on the dependency-pair method [1]. Moreover, in order to find reduction orders for ensuring termination, we have used the combination of polynomial interpretation and SAT solving proposed in [5]. We have considered the class of linear polynomial orderings with coefficients in $\{0, 1\}$ as the search space for the reduction orders. We refer to our implementation with this termination checker as MKBcs/POL. All experiments have been performed on a workstation equipped with Intel Xeon 2.13GHz CPU and 1GB system memory.

7.1 Comparison with naive parallelization

We have compared MKBcs with the naive parallelization approach, in which all processes are executed independently. In order to simulate the naive parallelization in the framework of MKBcs, we have applied the ORIENT rule only for a single process in the way described in 6.1. The results are summarized in Table 1, where "naive" columns show the results when the line marked by **(*)** in Fig. 6 is executed, and "MKB" columns show the results when the line is skipped. The "all" columns show the total time (in seconds) and "re/de" columns show the time consumed by REWRITE-1,2 and DEDUCE rules for each problem. Hyphens indicate that we could not get the results within 24 hours. We can see that the node-based rewriting and deducing is very effective, especially for the problems requiring a long computation time.

7.2 Comparison of MKBcs/AProVE with SLOTHROP

We have compared our approach with SLOTHROP, the first constraint-based completion tool described in [14]. SLOTHROP uses AProVE [6], one of the most powerful termination checkers known in the literature. In order

Table 1 Comparison with naive parallelization

Problem	naive		MKB	
	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

to compare MKBcs and 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. The results are summarized in Table 2, where "all" columns show the total time and "tc" columns show the number of termination checking. 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 depth-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 wrong. 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. Therefore, in principle, 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. Apart from the performance, the convergent rewrite systems generated by the two systems 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 Section 5 and (2) the cache-based condition checking described in Section 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 all described techniques 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.

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

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 the well-encoded bit string systems for representing 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 incorporating the dependency-pair method as the associated termination

checker. The experiments show that MKBcs is clearly more efficient than the simulation of naive parallelization. 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.

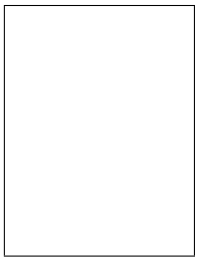
As future work, we are planning to incorporate the ideas of AC-completion and unfailing completion into MKBcs.

Acknowledgments

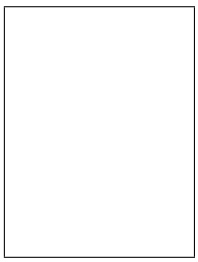
This work was partially supported by JSPS Grant-in-Aid for Scientific Research (C), No. 19500020.

References

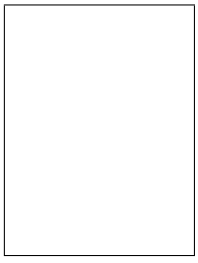
[1] T. Arts and J. Giesl, "Termination of term rewriting using dependency pairs", Theoretical Computer Science, vol. 236, pp. 133-178, 2000.
[2] F. Baader and T. Nipkow, Term Rewriting and All That, Cambridge University Press, 1998.
[3] L. Bachmair, Canonical Equational Proofs, Birkhäuser, 1991.
[4] N. Dershowitz, "Termination of rewriting", Journal of Symbolic Computation, vol. 3, issue 1-2, pp. 69-116, 1987.
[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] D. E. Knuth and P. B. Bendix, "Simple word problems in universal algebras", in J. Leech(ed.), Computational Problems in Abstract Algebra, pp. 263-297, Pergamon Press, 1970.
[8] M. Kurihara and H. Kondo, "Completion for multiple reduction orderings", Journal of Automated Reasoning, vol.23, no.1, pp. 25-42, 1999.
[9] N. Hirokawa and A. Middeldorp, "Dependency pairs revisited", Proc. 15th International Conference on Rewriting Techniques and Applications, vol. 3091 of Lecture Notes in Computer Science, pp. 249-268, 2004.
[10] 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.
[11] J. Steinbach and U. Kühler, "Check your ordering - termination proofs and problems", Technical Report SR-90-25, Universität Kaiserslautern, 1990.
[12] Terese, Term Rewriting Systems, Cambridge University Press, 2003.
[13] 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.
[14] 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.
[15] H. Zankl and A. Middeldorp, "Satisfying KBO constraints", Proc. 18th International Conference on Rewriting Techniques and Applications, vol. 4533 of Lecture Notes in Computer Science, pp. 389-403, 2007.



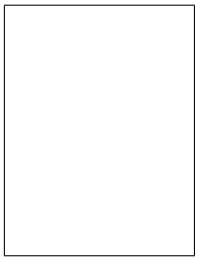
Haruhiko Sato



Masahito Kurihara



Sarah Winkler



Aart Middeldorp