Fast Knuth-Bendix Completion with a Term Rewriting System Compiler

Yoshihito TOYAMA
NTT Basic Research Laboratories
3-9-11 Midori-cho, Musashino-shi, Tokyo 180, Japan

Abstract

A term rewriting system compiler can greatly improve the execution speed of reductions by transforming rewriting rules into target code. In this report, we present a new application of the term rewriting system compiler: the Knuth-Bendix completion algorithm. The compiling technique proposed in this algorithm, is dynamic in the sense that rewriting rules are repeatedly compiled in the completion process. The execution time of the completion with dynamic compiling is ten or more times as fast as that obtained with a traditional term rewriting system interpreter.

1. Introduction

A term rewriting system compiler translates rewriting rules into target code in another language such as LISP [5, 11], PASCAL [2], C [10], or an assembly language [4]. The execution time of compiled code is usually hundreds or thousands of times as fast as that of the corresponding term rewriting system interpreter.

In this report, we propose a new application of a term rewriting system compiler: the Knuth-Bendix completion algorithm [7]. The Knuth-Bendix completion algorithm is well known as a useful technique to solve the word problem of an equational theory [3, 7]. However, as far as the author knows, the Knuth-Bendix algorithm has up to now only been executed with a term rewriting system interpreter. The reason why it has never been executed with a term rewriting system compiler is as follows:

(1) In the completion process, rewriting rules are repeatedly modified; hence, they must be recompiled each time. This dynamic compiling is difficult for most term rewriting system compilers which cannot produce compiled code quickly.

(2) Most term rewriting system compilers have been developed for functional programming languages or algebraic specifications of which rules are restricted by some properties, such as left-linearity, non-ambiguity (i.e. the non-overlapping property), and strong sequentiality [2, 4, 8, 10]. On the other hand, the completion process must treat any rules without such restrictions.

Recently, Tomura [11] and Kaplan [5] proposed independently an interesting term rewriting system compiler based on some tricky use of LISP features which translates rewriting rules into LISP functions; for example, the term rewriting system

$$R_{minus} \quad \begin{cases} minus(x, 0) \triangleright x \\ minus(s(x), s(y)) \triangleright minus(x, y) \\ minus(x, x) \triangleright 0 \end{cases}$$

is translated into the LISP function <u>*minus*</u> as follows:

```
(defun minus (X Y)
    (cond [(eq Y 0)    X]
          [(and (eq (car X) 's) (eq (car Y) 's))    (minus (cdr X) (cdr Y))]
          [(equal X Y)    0]
          [t    (list 'minus X Y)] )).
```

Their compiler has advantages for the Knuth-Bendix completion algorithm; it produces compiled code quickly and accepts a wide class of rewriting rules. Furthermore, since the completion process treats only terminating rules, we may use the innermost reduction strategy for computing normal forms which simplifies the compiler for our application. Thus, we can easily apply this compiler to the completion algorithm. Several benchmarks show that the execution time of the completion with dynamic compiling is ten or more times as fast as that obtained with a term rewriting system interpreter.

2. Benchmarks for TRS Compiler

The purpose of this benchmarks is to compare the performance of a traditional TRS (Term Rewriting System) interpreter and that of the TRS compiler proposed by Kaplan before using them in the Knuth-Bendix completion algorithm. Both systems compute normal forms in the same way, i.e. by the left-innermost reduction strategy [6, 8].

The original TRS compiler proposed by Tomura [11] and Kaplan [5] translates rewriting rules into compiled code through two phases; in the first phase, the TRS compiler generates a LISP function $\underline{f}$ for each function symbol f, and in the second phase, this function $\underline{f}$ is compiled into native machine code by a LISP compiler. However, the execution time of the second phase is usually too long for repeated compiling in the Knuth-Bendix completion process. Thus, our TRS compiler does not have the second phase; the LISP function $\underline{f}$ generated by the first phase is directly computed by a LISP interpreter.

By using the following rules, the factorial function $fact(n) = n!$ is computed by both systems. Here, natural numbers n are represented in the usual way: $0, s(0), s(s(0)), \cdots$.

$$R_{fact} \quad \begin{cases} plus(x,0) \triangleright x \\ plus(x,s(y)) \triangleright s(plus(x,y)) \\ times(x,0) \triangleright 0 \\ times(x,s(y)) \triangleright plus(times(x,y),x) \\ fact(0) \triangleright s(0) \\ fact(s(x)) \triangleright times(s(x),fact(x)) \end{cases}$$

The benchmarks have been made on a TOSHIBA J-3100GT (IBM PC compatible laptop computer with CPU 80286 (8 Mhz)). Both the TRS interpreter and the TRS compiler are written in MuLISP-87. The TRS interpreter is compiled by MuLISP-87 compiler. As stated above, the functions generated by the TRS compiler are not compiled by LISP compiler; they are directly evaluated by MuLISP-87 interpreter. The results are shown in the following table:

	n=0	n=1	n=2	n=3	n=4	n=5	n=6	n=7
T_I (sec.)	0.16	0.55	1.20	3.24	19.67	340.30	∞	∞
T_C (sec.)	0.00	0.00	0.01	0.02	0.07	0.22	1.10	7.30
T_I/T_C	—	—	120	162	281	1546.82	∞	∞

Here, T_I and T_C show the execution time by the interpreter and by the compiled code respectively. ∞ in T_I shows that the computation is impossible because of memory overflow.

3. Fast Knuth-Bendix Completion

3.1. Completion with Dynamic Compiling

A complete (i.e., confluent and terminating) term rewriting system R which produces the same equality as the one generated by an equational theory E is very important to solve the word problem of E [3, 7]. Knuth and Bendix [7] proposed a famous procedure to find a complete term rewriting system R from a given an equational theory E. According to Dershowitz [1] and Klop [6], a simple version of this procedure is given as follows:

Knuth-Bendix completion algorithm

The procedure has as input a finite set R of rules, a finite set E of equations, and a program to compute a well-founded monotonic ordering $>$ on terms. The critical pairs $\langle P, Q \rangle$ are presented in E as equations $P = Q$.

Repeat while E is not empty. If E is empty, we have successful termination.

(1) Remove an equation $M = N$ (or $N = M$) from E such that $M > N$. If such an equation does not exist, terminate with failure.

(2) Move each rule from R to E whose left-hand side contains an instance of M.

(3) Extend E with all critical pairs in R caused by the rule $M \triangleright N$.

(4) Add $M \triangleright N$ to R.

(5) Use R to normalize the right-hand sides of rules in R.

(6) Use R to normalize both sides of equations in E. Remove each equation that becomes an identical equation.

The Knuth-Bendix completion algorithm spends the greater part of the execution time on (i) computing normal forms and (ii) generating critical pairs. In particular, the completion procedure with a TRS interpreter spends most time on normalizing. Hence, the execution time of the completion can be extremely improved by replacing the TRS interpreter with the TRS compiler described in Section 2. The procedure with the TRS compiler is given as follows:

Knuth-Bendix completion algorithm with TRS compiler

Repeat while E is not empty. If E is empty, we have successful termination.

(1) Remove an equation $M = N$ (or $N = M$) from E such that $M > N$. If such an equation does not exist, terminate with failure.

(2) Move each rule from R to E whose left-hand side contains an instance of M.

(3) Extend E with all critical pairs in R caused by the rule $M \triangleright N$.

(4) Add $M \triangleright N$ to R.

(5) Compile R into compiled code C.

(6) Use C to normalize the right-hand sides of rules in R.

(7) Use C to normalize both sides of equations in E. Remove each equation that becomes an identical equation.

In the above algorithm, normal forms are computed with compiled code C in (6), (7). (On the other hand, a traditional algorithm computes normal forms with rewriting rules R in interpreter mode. See (5), (6) in the previous algorithm.) Thus, rewriting rules R are repeatedly compiled into compiled code C at (5) while the iteration continues. We call this *dynamic compiling*.

Remark. Purdom and Brown [9] also proposed a *dynamic updating* technique for the Knuth-Bendix completion algorithm different from our *dynamic compiling*. They showed that by repeatedly updating their pattern representation, the number of matching routine called in the completion process can be reduced into about 1/5.

3.2. Benchmarks for Completion with TRS Compiler

We compare the execution time of the Knuth-Bendix completion with dynamic compiling, with that of a traditional completion. The examples for the benchmarks are given in the appendix. The benchmarks have been made under the same conditions described in Section 2. Both the completion algorithms are written in MuLISP-87 and compiled by MuLISP-87 compiler. The results are shown in the following table:

	group	group'	group"	l-r-s	c-group	rev
T_I (sec.)	107.73	130.19	1440.30	222.40	5.27	10.50
T_C (sec.)	10.11	12.14	60.26	15.32	1.43	2.41
T_I/T_C	10.66	10.72	23.90	14.52	3.69	4.36
N	16	18	40	16	3	7

Here, T_I and T_C show the execution time by the completion algorithm with an interpreter and with dynamic compiling, respectively. N shows the number of the compiling times in the completion process. The benchmarks show that the execution time of the completion with dynamic compiling is ten or more times as fast as that with a traditional term rewriting system interpreter.

Acknowledgment

The author is grateful to Jan Willem Klop, Pierre Lescanne, and Stéphane Kaplan for helpful suggestions.

References

[1] N. Dershowitz, Computing with rewriting systems, *Information and Control 65* (1985) 122-157.

[2] A. Geser, H. Hussmann, and A. Mück, A compiler for a class of conditional term rewriting systems, *Lecture Notes in Comput. Sci. 308* (Springer-Verlag, 1988) 84-90.

[3] G. Huet and D.C. Oppen, Equations and rewrite rules: a survey, in: R.V. Book, ed., *Formal languages: perspectives and open problems*, (Academic Press, 1980) 349-405.

[4] K. Inoue, H. Seki, K. Taniguch, and T. Kasami, Compiling and optimizing methods for the functional language ASL/F, *Science of Computer Programming 7-3* (1986) 297-312.

[5] S. Kaplan, A compiler for conditional term rewriting systems, *Lecture Notes in Comput. Sci. 256* (Springer-Verlag, 1987) 25-41.

[6] J. W. Klop, Term rewriting systems: a tutorial, *EATCS Bulletin 32* (1987) 143-182.

[7] D.E. Knuth and P.G. Bendix, Simple word problems in universal algebras, in: J. Leech, ed., *Computational problems in abstract algebra* (Pergamon Press, 1970) 263-297.

[8] M. J. O'Donnell, *Equational logic as a programming language*, (The MIT Press, 1985).

[9] P. W. Purdom and C. A. Brown, Fast many-to-one matching algorithms, *Lecture Notes in Comput. Sci. 202* (Springer-Verlag, 1985) 407-416.

[10] M. Sakai, T. Sakabe, and Y. Inagaki, Direct implementation system of algebraic specifications of abstract data types, *Computer Software 4-4* (Iwanami, 1987) 16-27, *in Japanese.*

[11] S. Tomura and K. Futatsugi, Transformation system from term rewriting systems into LISP programs, *IEICE (the Institute of Electronics, Information and Communication Engineers) technical report SS86-11* (1986) 15-20, *in Japanese.*

Appendix

We present the equational theories E [3, 7] used in the benchmarks of Section 3.2.

Groups (group). E_{group} defines group theory by:

$$E_{group} \quad \begin{cases} 1 * x = x \\ I(x) * x = 1 \\ (x * y) * z = x * (y * z) \end{cases}$$

Groups'(group'). $E_{group'}$ defines group theory by a right identity and a right inverse:

$$E_{group'} \quad \begin{cases} x * 1 = x \\ x * I(x) = 1 \\ (x * y) * z = x * (y * z) \end{cases}$$

Groups" (group"). $E_{group''}$ defines group theory by Taussky's presentation:

$$E_{group''} \quad \begin{cases} 1 * 1 = 1 \\ x * I(x) = 1 \\ (x * y) * z = x * (y * z) \\ f(1, x, y) = x \\ f(x * y, x, y) = g(x * y, y) \end{cases}$$

(l,r)-Systems (l-r-s). E_{l-r-s} defines (l,r)-system by:

$$E_{l-r-s} \quad \begin{cases} 1 * x = x \\ x * I(x) = 1 \\ (x * y) * z = x * (y * z) \end{cases}$$

Central Groupoids (c-group). $E_{c-group}$ defines central groupoids theory by:

$$E_{c-group} \quad \left\{ \ (x * y) * (y * z) = y \right.$$

Reverse (rev). E_{rev} defines the reverse of a list:

$$E_{rev} \quad \begin{cases} append(nil, y) = y \\ append(cons(x, y), z) = cons(x, append(y, z)) \\ reverse(nil) = nil \\ reverse(cons(x, y)) = append(reverse(y), cons(x, nil)) \\ reverse(reverse(x)) = x \end{cases}$$