Cybernetics and Systems Analysis, Vol. 38, No. 5, 2002

CYBERNETICS

A REWRITING MACHINE AND OPTIMIZATION OF STRATEGIES OF TERM REWRITING

A. A. Letichevskii[a] **and V. V. Khomenko**[b]

UDC 623/518.3/517.5

An algebraic specification of a new rewriting machine for fast rewriting of terms is considered. Theorems on the correctness of this specification are proved. A method for optimization of a strategy of iterative rewriting is proposed.

Keywords: *term rewriting, rewriting machine, algebraic programming, algebraic specification, optimization of programs.*

Currently, the paradigm of algebraic programming becomes one of the most important paradigms of declarative programming. Algebraic programming is based on methods of term rewriting [1], which are widely used independently and in programming systems based on other paradigms (in functional, logical, object-oriented, and agent-based programming, in computer algebra and systems of theorem proving). Among systems of algebraic programming that appeared in recent years, noteworthy are Maude, Elan, Cafe-obj, and APS [2–7]. The theory of rewriting systems is developed in detail and is presented, for example, in [8, 9].

The system of algebraic programming (APS) was created in V. M. Glushkov Cybernetics Institute of the National Academy of Sciences of Ukraine in the early nineties. The system is a professionally oriented tool for developing applied systems and is based on algebraic and logic models of object domains. The basic means of programming are systems of rewriting rules. The algebraic programming language APLAN used in the system integrates fundamental paradigms of programming and its semantics can be easily extended with the help of metaprogramming, which is also based on rewriting.

Fast term rewriting is conditioned by the use of efficient parallel matching for realization of the rewriting machine considered in this article. The corresponding program was developed by S. V. Konozenko and was realized in the language C in the first version of APS.

This article deals with a new version of the rewriting machine developed for a new version of the APS system. The machine is represented in the language APLAN used in the capacity of the language of executable specifications. Such a representation allows one to construct the proof of correctness of the corresponding program, to quickly develop a reliable implementation based on the programming language (C++) being used, and to obtain new modifications of the machine that support various special rewriting strategies.

1. THE ALGEBRAIC PROGRAMMING SYSTEM APS

The main distinctive feature of APS is the fact that the use of systems of rewriting rules can lean upon various strategies of rewriting. This approach allows one to consider not only canonical (confluent and Noetherian) but also any other systems of rewriting rules, and algebraic programs can be constructed as combinations of systems of rules with various strategies of using them.

---

[a]Cybernetics Institute, National Academy of Sciences of Ukraine, Kiev, Ukraine, *al@letichevsky.kiev.ua*. [b]Taras Shevchenko University, Kiev, Ukraine. Translated from Kibernetika i Sistemnyi Analiz, No. 5, pp. 3-17, September-October, 2002. Original article submitted November 12, 2001.

Another distinctive feature of APS is the possibility of combining the imperative and algebraic programming methods. The input language (APLAN) of the system allows one not only to describe an algebraic environment (components, operations, and identities of a data algebra) but also to write procedures and to call them from algebraic programs (systems of rewriting rules). Convenient tools for representation of procedures allow the user to manipulate standard strategies and those developed by him for applying rewriting rules. Considering algebraic expressions (terms) up to some congruence relations, rewriting strategies can use various properties of a data algebra. To this end, the mechanism of canonical (normal) forms of algebraic expressions is used. This mechanism operates whenever some rewriting rule is applied. The choice of canonical forms is determined by the requirements of an object domain and is controlled by the user. A more complete description of application of systems of rewriting rules and computational strategies can be found in [5–7].

2. SYSTEMS OF REWRITING RULES

In APS, the syntax of systems of rewriting rules is as follows:

```
<system of rewriting rules>:: = rs(<list of variables>)
   (<list of rules separated by points>)
<rule>::=<simple rule> | <conditional rule>
<simple rule>::=<algebraic expression> =
     <algebraic expression>
<conditional rule>::=<condition>–>(<simple rule>)
<variable>::=<identifier>
```

In APS, rewriting strategies are based on two basic internal procedures applr and appls. The operator applr(t, R) tries to apply a rule of a system R to a term t. If applicable rules are absent, then the variable yes assumes the value 0; otherwise, the first applicable rule is applied and the variable yes assumes 1. A simple rule is applied as usual, namely, the term is matched with the left side of the rule and if the result is positive, then the term is replaced by the right side of the rule and the substitution of its variables is made. Then the term obtained is reduced to its canonical form with the help of rules similar to the rules of computation of the value of a term but without replacing the values of names. The application of a conditional rule begins with matching. If the result is positive, then the condition is reduced to its basic canonical form by the canonizing function CAN. If the result equals 1, then the rule will continue to be applied as usual; otherwise, the application is canceled.

In the new version of APS, the semantics of application of conditional rewriting rules is changed as follows: in computing a condition, the variables matched are protected from reduction to canonical forms. The authors' experience suggests that this semantics is more convenient, and the compatibility with the previous semantics can be obtained by the explicit call of the function CAN.

3. STRATEGIES OF REWRITING

A strategy can be defined as a set of allowable histories of rewriting for a given set of rules. In order that a rewriting strategy be constructive, it should be defined with the help of an algorithm that recognizes or generates allowable histories of rewriting. The possibility of using a wide spectrum of various types of strategies renders the language of algebraic programming flexible, convenient, and expressive. In practice, of great interest are local computational strategies that can be defined with the help of answers to the questions given below.

- What is the order in which the subexpressions of a given expression are observed and the applicability of rules is checked?
- What is the order in which the rules are observed?
- What is the continuation of the process after successful completion of the preceding step of rewriting?
- What are the conditions of termination of rewriting?

There are the following two main answers to the first question: the top-down search (the outermost occurrence) and bottom-up search (the innermost occurrence). These answers correspond to calls by name and by value in functional programming.

The simplest answer to the second question is the examination of relationships according to some order; the use of the next relationship depends on the preceding one and on whether it is possible to use it in principle. After successful application of some relationship, it can be applied to the same subexpression until it is possible, and then the search is being continued in the same or opposite direction. Another possibility is to begin with the very outset whenever some rule can be applied (as in Markov algorithms).

After completion of the complete traversal of an expression, the rewriting process can be terminated (a single-pass application) or is continued in the same or opposite direction (a repeated application). An expression is called normalized if none of the rules is applicable to its subexpressions. A strategy is called normalizing if it comes to an end only after obtaining a normalized expression. A wide class of local strategies can be defined by the representation of recursive programs in the APLAN language of APS. Some basic rewriting strategies used in this system (all of them are ultimately reduced to the repeated use of the basic strategy applr in various contexts) are as follows:

```
appls:=proc(t, p)loc(Yes)(
        applr(t, p);
        Yes:=yes;
        while(yes,
            applr(t, p)
        );
        yes: = Yes
);

ntb:=proc(t, R)loc(s, i)(
        appls(t, R);
        forall(s=arg(t, i),
            ntb (s, R)
        );
        t:=can (t)
);

nbt:=proc(t, R)loc (s, i)(
        forall(s=arg (t, i),
            nbt (s, R)
        );
        appls(t, R);
        t:=can(t)
);

can_ord:=proc(t, R1, R2)loc(s, i)(
        t:= can(t);
        appls(t, R1);
        forall(s=arg(t, i),
            can_ord(s, R1, R2)
        ); can_up(t, R2)
);

can_up:=proc(t, R)loc(s, i)(
        appls(t, R);
        while(yes,
            forall(s=arg(t, i),
                can_up(s, R)
            );
            appls (t, R)
        );
        t:=can (t);
        merge (t)
);
```

The procedure appls applies a system R to a term t until it is possible and produces the result $\text{yes}=1$ if the application is successful; nbt is a single-pass bottom-up strategy and ntb is a single-pass top-down strategy. The strategy $\text{can\_ord}(t, R, S)$ traverses the tree representing a term from the top down and from left to right. After the first arrival at a node in traversing from above, it applies a system R and, after the complete traversal of all the daughter nodes and return to the node, a system S is applied. If the application of S is successful, then S is repeatedly applied to all the daughter nodes. After complete processing of each node, the corresponding terms are ordered with respect to associative-commutative operations (the function merge merges the terms that are already ordered).

4. A REWRITING MACHINE

The rewriting machine (REM) being considered realizes the rewriting strategy applr. This strategy is fundamental and, based on it, all the other strategies are specified. Therefore, the efficiency of its realization is of paramount importance. Substituting the strategy applr in other strategies and performing appropriate transformations, one can obtain specialized rewriting machines for these strategies.

To obtain an efficient realization of the strategy applr, the considerations presented below are used.

• If the left sides of several adjacent rules are identical, then they can be matched only once (in some cases, this kind of optimization can also be used for nonadjacent rules; the conditions under which two rules can be interchanged are considered below).

• We need not consider the rules in which the main operation of the left side differs from the main operation of the term being processed (i.e., rewriting rules can be grouped according to the main operations of their left sides). Similar considerations also hold for the main operations of subterms of the left sides of rules.

For realization of the above considerations, the rewriting machine REM uses a special syntax for representation of systems of rewriting rules, namely, the input language REM.

4.1. The Syntax of the REM Language. The syntax of the language of the rewriting machine is expressed by the following parametric grammar:

```
<closed REM-program>::=
        <program heading> <program of rank 1>
<program heading>::=Rs array (<list of underlines>)
<list of underlines>:: = _ | _, <list of underlines>
<REM-program>::=<program of rank k>
<program of rank k>::=<protected program of rank k> |
         <protected program of rank k> + <program of rank k>
<protected program of rank 0>::=rewrite(<term>) |
        If(<term>, rewrite(<term>))
protected program of rank n>::=match(<term>)
         <program of rank n-1> |
         test(<m-type>) <program of rank n+m-1>
```

Here, m and n are positive integers, k is a nonnegative integer, <term> is a term (algebraic expression) of the APLAN language, $<m$-type> is a term of the form $\omega((),\ \ldots,())$, where ω is a label of arity m. The terms of the form $\mathrm{var}(i)$, where i is a positive integer that does not exceed the number of underlines in the program heading, and only such terms are considered as variables of closed programs. Programs of rank k are components of completely generated programs of rank 1. The ranks of the corresponding components should be balanced just as the components of a term are balanced in a parenthesis-free notation. As will be shown below, each closed REM-program uniquely specifies some system of rewriting rules of the APLAN language (up to variable renaming).

4.2. Algebra of REM-Programs. Let $T_\Omega(Z)$ be the basic algebra of terms of some algebraic program. A set Ω is the signature of operations of this algebra, and a set Z is the generating set of terms of arity 0. REM-programs form a multisort algebra $R=(R_k)_{k\in N}$ over $T_\Omega(Z\cup V)$, where V is the set of variables of these programs. Here, R_k is the set of programs of rank k. The operations of the algebra being considered are as follows:

(1) $+: R_k \times R_k \to R_k,\ k>0$;
(2) $\mathrm{test}(\omega((),\ \ldots,())): R_{n+m-1} \to R_n,\ m>0,\ n>0,\ \omega\in\Omega,\ m=\mathrm{ART}(\omega)$;
(3) $\mathrm{match}(t): R_n \to R_{n+1},\ n\ge 0,\ t\in T_\Omega(Z\cup V)$;
(4) $\mathrm{rewrite}(t)\in R_0,\ t\in T_\Omega(Z\cup V)$;
(5) $\mathrm{If}(u,\mathrm{rewrite}(t))\in R_0,\ u,t\in T_\Omega(Z\cup V)$.

The operations of the form (4) and (5) have the zero arity and are generating elements of the algebra of REM-programs. As follows from the definitions, any program of rank k can be represented in the form of an expression of the algebra of REM-programs.

Definition. We call a REM-program a **match**-program if an operation of the form $\mathrm{test}(\omega((),\ \ldots,()))$ is not used in it and an **elementary** program if the operation + is not used in it.

Elementary match-programs of rank 1 are of the form

$$\text{match } t \text{ rewrite } t'$$

or of the form

$$\text{match } t \text{ If}(s, \text{rewrite} t').$$

Let $P = \text{Rs}\, S_m Q$ be a closed REM-program, where S_m is an array consisting of m underlines, Q is an elementary match-program of rank 1, and $V = (\text{var}(1), \ldots, \text{var}(m))$. Then the program is a system of the form $\text{rs}\, V(t = t')$ in the first case and a system of the form $\text{rs}\, V(s -> (t = t'))$ in the second case. An arbitrary match-program of rank 1 is the "sum" of elementary programs. If Q is an arbitrary match-program of rank 1, then P must be equivalent to the system of rewriting rules corresponding to the elementary component programs of the program Q. Based on such a correspondence, an arbitrary system of rewriting rules can be easily translated into a REM-program with the help of the following simple translator (the program heading is assumed to be already translated):

```
equ2match:=rs(u, s, t, p, q, r)(
    (s = t) = match(s)rewrite(t),
    (u–>(s = t)) = match(s)If(u, rewrite(t)),
    (p, q) = equ2match p + equ2match q
);
```

The expressions

$$\text{match}(t)$$

$$\text{rewrite}(t)$$

$$\text{If}(u, \text{rewrite}(t))$$

$$\text{test}(\omega((), \ldots ()))$$

form the system of commands of the rewriting machine. Let us consider the meaning of these commands. The command match matches the input term with a sample t. The command rewrite realizes rewriting, and If is the command of conditional rewriting of the input term in a new term t. The command test checks whether the main operation of the input term is the operation ω.

4.3. Equivalence of REM-Programs. Let us consider the following system of identities on the set of REM-programs:

$$(p + q) + r = p + (q + r) \tag{1}$$

$$\text{match}(t)(p + q) = \text{match}(t)p + \text{match}(t)q \tag{2}$$

$$\text{test}(t)(p + q) = \text{test}(t)p + \text{test}(t)q \tag{3}$$

$$\text{test}(\omega((), \ldots, ()))\text{match}(t_1)\ldots\text{match}(t_m)p = \text{match}(\omega(t_1, \ldots t_m))p, \tag{4}$$

where $p, q,$ and r are arbitrary programs of the same kind in each identity and $m = \text{ART}(\omega(\ldots))$. Identities (1)-(4) determine a congruence relation ~ on the set of REM-programs, which is called the syntactic equivalence. If identities (1)-(4) are considered as a system of rewriting rules, then we can see that it has the important properties considered below.

- In the system, all the rules are left-linear and critical pairs are absent, i.e., the system is regular and, hence, it is confluent.
- The system of rewriting rules is Noetherian.

These properties and the Knuth-Bendix theorem immediately imply the existence and uniqueness of the canonical form of any REM-program with respect to these identities. We can now prove the following result.

THEOREM. Let p be some REM-program of rank k, and let p' be its canonical form. Then p' is of the form $\sum_{i=1}^{n} p_i$, where p_i are elementary match-programs of rank $k, i = 1, \ldots, n$.

To prove the theorem, we note that any REM-program can be transformed with the help of equalities (1)–(3) into the sum of elementary programs. Next, with the help of relationship (4), all the occurrences of the operation test can be eliminated from each elementary program, provided that its rank is preserved. In the APLAN language, the algorithm of reduction of a REM-program to its canonical form is of the form

```
NAMES  t2m_up,  test2match;
test2match:=rs(t,  p,  q)(
    match(t)p  =  t2m_up  match(t)  test2match  p,
    test(t)p  =  t2m_up  test(t)  test2match  p,
    p  +  q  =  t2m_up  (test2match  p  +  test2match  q)
);
NAME  Starg;
t2m_up:=rs(t,  p,  q,  r)(
    (p+q)+r  =  t2m_up  p+t2m_up(q  +  r),
    match(t)(p  +  q)  =  match(t)p+t2m_up  match(t)q,
    test(t)  (p  +  q)  =  Starg(ART(t),  1,  t,  p)+t2m_up  test(t)q,
    test(t)  p  =  Starg(ART(t),  1,  t,  p)
);

Starg:=proc(x)(
    appls(x, Starg_rs);
    return  x
);
Starg_rs:=  rs(m,  i,  t,  p,  q)(
    (m,  m,  t,  match(p)q)  =  match(starg(t,  m,  p))q,
    (m,  i,  t,  match(p)q)  =  (m,  i  +  1,  starg(t,  i,  p),  q)
);
```

The function starg(t, i, p) sets the ith argument of the main operation of a term t to p.

4.4. The Algorithm of the Rewriting Machine. As follows from the theorem, a system of rewriting rules can be assigned to each closed REM-program. This implies the following formal requirement on the algorithm of functioning of the rewriting machine: the algorithm must calculate a function of two arguments. The first argument is the term to be rewritten and the second argument is a rewriting program (REM-program). The result to be obtained must be the rewriting of the initial term by application of the corresponding system of rewriting rules to it. In other words, it is required to write a procedure napplr(t, p) that transforms a term t in the same way as applr(t, R), where R is the rewriting system corresponding to a program p. The upper level of the program napplr is of the form

```
NAME  napplr; NAME  appl;
napplr:=proc(t,p)loc(pr,Yes,s)(
    let(p,  Rs  pr  p);
    s–>(p,t  Nil,pr)+Nil;
    appls(s,  appl);
    let(s,1:s);
    yes–>t:=  s
);
```

The basic part is represented by a system of rewriting rules appl, which is iteratively applied to a state s of the rewriting machine. The initial state contains a program p of rank 1, the initial term t with the constant Nil applied to it, and an array pr consisting of underlines. This is the precondition specifying the requirements on the operator appls$(s,$ appl$)$. The postcondition is as follows. If the system R corresponding to the program p is applicable to the term t, then, as a result of rewriting, we have $s = (1{:}R(t))$. Otherwise, we have $s = 0$.

Let us consider the system of rules appl given below.

```
NAME  perform;
appl:=rs(p,q,r,t,pr)(
        (1:  t  )  +  r  =  (1:t),
        (p  +  q,  t,  pr)  +  r  =  (p,t,new(pr))  +  (q,t,pr)  +  r,
        (p,  t,  pr)  +  r  =  perform(p,t,pr)+  r
);
NAMES  vsc,  do_match,  check_match,  concline;
perform:=rs(p,q,s,t,pr)(
        (rewrite(q),  t,pr)  =  (1:vsc(q,pr)),
        (match(p)q,  s  t,pr)  =  check_match(q,t,do_match(p,s,pr)),
        (test(p)q,  s  t,pr)  =  is_type(p,s)&(q,concline(s,t),pr),
        (If(p,q),  t,pr)  =  (vsc(p,pr)==1)&(q,t,pr)
);
check_match:=rs(p,t)(
        (p,  t,  0)  =  0
);
```

The algebraic program appl uses the auxiliary functions listed below.

• The function do_match(p, s, pr) has the following arguments: p is a term that can depend on variables (var$(1), \ldots,$ var(m)), s is a constant term, and pr is an m-dimensional array of already determined values of the variables. If the value of a variable is not determined, then it is assumed to be an underline. We denote by $Sub(p, pr)$ the result of substitution of already determined values of the variables in p. The term t is matched with the sample $Sub(p, pr)$. If this matching succeeds, then a new (extended) value of pr is returned; otherwise, the value 0 is returned.

• The function is_type(p, s) determines whether the main operations of terms p and s coincide.

• The procedure vsc(p, pr) reduces a term p to its basic canonical form with the help of the function can together with the substitution of the values of variables from the array pr in it. It realizes the final part of rewriting and, hence, it must be realized according to the requirements formulated in [7]. In particular, the semantics of the application operation and that of the operation ‘() that cancels the use of canonization must be taken into account.

• The function concline$(\omega(t_1, \ldots, t_m), t)$ returns the term $(t_1 t_2 \ldots t_m t)$.

To prove the correctness of the operator appls$(s,$ appl$)$, where

$$s = (p, t, \text{Nil}, pr)$$

satisfies the precondition for the initial state, we will use the theorem and prove the correctness by induction on the number of rewriting steps required for the reduction of the program p to its canonical form.

The basis of induction is easily checked. Let $p \sim p'$, i.e., p is transformed into p' in one step, and let, for p', the correctness be proved. Considering different cases of rewriting rules and also the cases where the result of computation of the function do_match is equal to zero, we obtain that, after a finite number of steps, the result of rewriting the state $s = (p, t\,\text{Nil}, pr)$ by the system appl coincides with the result of rewriting the state $s = (p', t\,\text{Nil}, pr)$ for which the induction hypothesis is fulfilled. Noting that this statement also implies the termination of the process of rewriting, we obtain the proof of the complete correctness of the operator appls$(s,$ appl$)$.

The functions that are realized in the APLAN language and are used in the algorithm of the rewriting machine are presented below without any substantiation.

```
NAME  Pr;
Pr:=Nil;
vsc:=proc(t,pr)loc(opr)(
        opr:=Pr;
        Pr–>new(pr);
        t–>vsc_rec(t);
        Pr–>opr;
        return  (t)
);
vsc_rec:=proc(t)loc(i,j,r,s,pr,p)(
        let(t,var(i));
        yes–>return  Pr(i);
        is_type(t,  ‘(‘(Nil))  )–>(
            t–>vs  arg(t,1);
            return(t)
        );
        let(t,p  s);
        yes–>(
            p–>is_rs(p);
            ~(equ(p,0))–>(
                t–>vsc_rec(s);
                napplr(t,p);
                return(t)
            )
        );
        t–>new(t);
        for(i:=1,i<=ART(t),i:=i+1,
                s–>arg(t,i);
                let(s,var(j));
                yes–>arg(t,i)–>Pr(j)
                else  arg(t,i)–>vsc_rec(s)
        );
        t–>can(t);
        return(t)
);
vs:=proc(p)loc(i,s)(
        let(p,var(i));
        yes–>return  Pr(i);
        p–>new(p);
        for(i:=1,  i  <=  ART(p),  i:=i+1,
                s–>arg(p,i);
                is_var(s)–>
                     arg(p,i)–>arg(Pr,arg(s,2))
                else  arg(p,i)–>vs(s)
        );
        return(p)
);
is_rs:=proc(t)(
        let(t,Rs  _);
        yes–>return  t;
        isname(t)–>return  is_rs  vl(t);
        return  0
);
do_match:=proc(l,t,pr)loc(opr)(
        opr:=Pr;
        Pr–>new(pr);
        match_rec(l,t)–>(
             pr–>Pr;
             Pr–>opr
        )else(
             pr–>0;
             Pr–>opr
        );
        return  pr
);
match_rec:=proc(l,t)loc(i,npr)(
        let(l,var(i));
        yes–>(
             equ(Pr(i),  _  )–>(
                     Pr(i)–>t;
                     return  1
             );
             return  equ(Pr(i),t)
        );
        is_type(t,l)–>(
             for(i:=1,i  <=ART(l),i:=i+1,
                npr–>match_rec(arg(l,i),arg(t,i));
                equ(npr,0)–>return  0
             );
             return  1
        );
        return  0
);
NAME  Con;
concline:=proc(s,p)loc(m,i)(
        m:=ART(s);
        for(i:=m,  i>0,  i:=i-1,
                p–>Con(arg(s,i),p)
        );
        return  p
);
Con:=rs(x,y)(
        (x,y)  =  ‘(x  y)
);
```

5. OPTIMIZATION OF REM-PROGRAMS

A system of rewriting rules can be represented by many different ways in the form of a REM-program. Therefore, the problem of improving its code arises, i.e., the problem of construction of a REM-program with the smallest possible execution time.

Different programs have different execution times that are equal to the sums of the execution times of their commands. Note that the time complexity of the command test(...) is bounded and the complexity of the command match(t) is proportional to the size of a term t. Unfortunately, the canonical form of a REM-program is frequently not optimal in the sense of execution time. As an example, let us consider the following system of rewriting rules that realizes fast exponentiation:

```
pw:=rs(x,n)(
        x^0=1,
        x^1=x,
        x^2=x*x,
(isnum(n)&((n mod 2)==0))–>(
        x^n=pw(pw(x^(n/2))^2)
),
isnum(n)–>(
        x^n=x*pw(x^(n-1))
)
);
```

After translation with the help of aplan2rem, we obtain the following REM-program:

```
pw:=Rs array(_,_)(
    match(var(1)^0)rewrite(1)
    +match(var(1)^1)rewrite(var(1))
    +match(var(1)^2)rewrite(var(1)*var(1))
    +match(var(1)^var(2))
        If(isint(var(2))&(var(2) mod 2==0),
        rewrite(pw(pw(var(1)^(var(2)/2))^2)))
    +match(var(1)^var(2))
        If(isint(var(2)),
            rewrite(var(1)*pw(var(1)^(var(2)-1))))
);
```

It is easy to see that this program is in its canonical form (REM-programs that are obtained as a result of functioning of aplan2rem are always in their canonical forms!). It successively examines (before the first application) all the rules of the system and, hence, is inefficient. An optimized version of this program is as follows:

```
pw:=Rs array(_,_)(
    test(NIL ^ NIL)
        match(var(1))(
            match(0)rewrite(1)
            +match(1)rewrite(var(1))
            +match(2)rewrite(var(1)*var(1))
            +match(var(2))(
                If(isnum(var(2))&(var(2) mod 2==0),
                    rewrite(pw(pw(var(1)^(var(2)/2))^2)))
                +If(isnum var(2),
                rewrite(var(1)*pw(var(1)^(var(2)-1))))
            )
        )
);
```

To obtain optimized programs, an optimizer realized in the form of the following APLAN-program is used:

```
NAMES split_atom,split_type,split_arg,m2t;
match2test:=rs(p,q,i,s)(
        match(var(i))p + q = m2t split_atom (match(var(i))p + q),
(ART(‘(s))==0)–>(
        match(s)p + q = m2t split_atom (match(s)p + q)
),
        match(s)p + q = m2t split_type (match(s)p + q)
);
```

```
m2t:=rs(s,p,q)(
        p+q  =  m2t  p  +  match2test  q,
        match(s)p  =  match(s)match2test  p,
        test  (s)p  =  test  (s)match2test  p
);

split_atom:=rs(p,q,r,s)(
    match(s)p  +  match(s)q  +  r  =
        split_atom(match(s)turn_right(p  +  q)  +  r),
    match(s)p  +  match(s)q  =
        match(s)turn_right(p  +  q)
);

NAME  turn_back;
split_type:=rs(p,q,r,s,t,u,v,i)(
/*  match  */
    match(s)p  +  match(s)q  +  r  =
        split_type(match(s)turn_right(p+q)  +  r),
match(s)p  +  match(s)q  =
        split_type(match(s)turn_right(p+q)),

    match(s)p  +  match(var(i))q  +  r  =
        match(s)p  +  match(var(i))q  +  r,
    match(s)p  +  match(var(i))q  =
        match(s)p  +  match(var(i))q,

is_type(‘(t),’(s))–>(
    match(s)p  +  match(t)q  +  r  =
        split_type(test(type(s))split_arg(s,p)+match(t)q  +  r)
),
is_type(‘(t),’(s))–>(
    match(s)p  +  match(t)q  =
        split_type(test(type(s))split_arg(s,p)+match(t)q)
),
match(s)p  +  match(t)q  +  r  =
    turn_back(match(t)q+split_type(match(s)p  +  r)),

/*  test  */
    test(s)p  +  match(var(i))q  +  r  =
        test(s)p  +  match(var(i))q  +  r,
    test(s)p  +  match(var(i))q  =
        test(s)p  +  match(var(i))q,
is_type(‘(t),’(s))–>(
    test(s)p  +  match(t)q  +  r  =
        split_type(test(s)turn_right(p+
        split_arg(t,q))+r)
),
is_type(‘(t),’(s))–>(
    test(s)p  +  match(t)q  =
        test(s)turn_right(p+split_arg  (t,q))
),
    test(s)p  +  match(t)q  +  r  =
        turn_back(match(t)q  +  split_type(test(s)p  +  r))
);

turn_back:=rs(t,p,q,r,s)(
        match(t)p+test(s)q+r  =  test(s)q+match(t)p+r,
        match(t)p+test(s)q  =  test(s)q+match(t)p,
        match(t)p+match(s)q+r  =  match(s)q+match(t)p+r,
        match(t)p+match(s)q  =  match(s)q+match(t)p
);
```

The function split_arg($f(t_1, \dots, t_m)$, p) returns match(t_1) … match(t_m) p.

To substantiate the correctness of the optimizer, the identities given below must be added to the relation of the syntactic equivalence ~ specified in Sec. 4.3.

- The operation + is associative:

$$(p1 + p2) + p3 \sim p1 + (p2 + p3). \qquad (5)$$

- match(t) and test(t) are additive operations, i.e., we have

$$\mathsf{match}(s)\,p+\mathsf{match}(s)q \sim \mathsf{match}(s)(p+q) \qquad (6)$$

$$\mathsf{test}(s)\,p+\mathsf{test}(s)q \sim \mathsf{test}(s)(p+q)\,.$$

- The decomposition rule is as follows:

$$\mathsf{match}(\omega(t_1,\dots,t_m))\,p \sim \mathsf{test}(\omega((),\dots,()))\mathsf{match}(t_1)\dots\mathsf{match}(t_m)\,p, \qquad (7)$$

where ω is an m-ary operation of the algebra $T_\Omega(Z)$.

- The commutativity rule is as follows: if t and s cannot be unified, then we have

$$\mathsf{match}(t)\,p+\mathsf{match}(s)q \sim \mathsf{match}(s)q+\mathsf{match}(t)\,p \qquad (8)$$

$$\mathsf{match}(t)\,p+\mathsf{test}(s)q \sim \mathsf{test}(s)q+\mathsf{match}(t)\,p$$

$$\mathsf{test}(t)\,p+\mathsf{match}(s)q \sim \mathsf{match}(s)q+\mathsf{test}(t)\,p$$

$$\mathsf{test}(t)\,p+\mathsf{test}(s)q \sim \mathsf{test}(s)q+\mathsf{test}(t)\,p$$

Comment 1. It follows from the associativity of the operation + that

$$p1+p2 \sim \mathsf{turn\_right}(p1+p2),$$

where the function turn_right (in this case) narrows the brackets for the upper sums to the right.

Comment 2. Relationships (6) and (7) coincide with the rules of syntactic equivalence. They imply the statement formulated below.

COROLLARY. If ART$(t)\neq 0$, then we have

$$\mathsf{match}(t)\,p \sim \mathsf{test}(\mathsf{type}(t))\mathsf{split\_arg}(t,\,p).$$

Comment 3. In the last three cases of the commutativity rule, the check of the possibility of unification is reduced to the call of the function is_type that checks the equality of the main labels of its arguments.

In what follows, to substantiate the correctness of the optimizer, it suffices to show that the resulting program is syntactically equivalent to the input one. To do this, a simple check of the fact that each rewriting rule retains the syntactic equivalence. In this case, it is necessary to take into account the following facts:

- in the right side, the application operation is interpreted only if its first argument is the name of a system of rewriting rules;
- the operation + is nowhere interpretable.

The proof of finiteness of the algorithm is carried out by induction on the construction of the term of a REM-program and presents no special problems.

6. OPTIMIZATION OF THE STRATEGY appls

The strategy appls consists of iterations of the strategy applr and can be specified by the following procedure:

```
appls:= proc(t, p)loc (Yes)(
    applr (t, p);
    Yes:= yes;
    while(yes,
        applr(t, p)
    );
    yes: = Yes
);
```

The above optimization of applr automatically increases the efficiency of appls, but some additional possibilities of optimization still remain. In fact, some information on the structure of the term being processed remains after each iteration and can be used at the succeeding iterations.

Example. Simulation of functioning of a finite-state automaton recognizing the representation of a natural number.

The corresponding system of rewriting rules is of the form

```
AUT:=rs(q,h,t,x)(
        (q0, "+", t) = (q1, t),
        (q0, "-", t) = (q1, t),
isnum(h)–>
        ((q0, h, t) = (q2, t)),
isnum(h)–>
        ((q0, h ) = 1),
isnum(h)–>
        ((q1, h, t) = (q2, t)),
isnum(h)–>
        ((q1, h) = 1),
isnum(h)–>
        ((q2, h, t) = (q2, t)),
isnum(h)–>
        ((q2, h) = 1),
        (q, x) = 0
);
```

If the seventh rule was applied at some iteration, then any of the preceding six rules cannot obviously be applied at the next iteration. This observation can be generalized as follows: if, at some iteration, a rule *r* is applied, then, at the next iteration, only the rules whose left sides are unified (in finite terms) with the right side of *r* can be applied.

If the right side of *r* contains interpreted operations, then, in general, the form of the resulting term can differ from the expected form. Therefore, before unification, all the subterms in the right side of *r* that are of the form $f(t_1, \dots, t_n)$, where f is an interpreted operation, must be replaced by a new (free) unification variable.

The idea of optimization is as follows:

(1) a system of rewriting rules is modified so that, after application of a rule, not only the resulting term but also the number of this rule is returned;

(2) systems R_i, $i = 1, \dots, n$, are constructed that consist of exactly the rules that can be applied after using the *i*th rule;

(3) after each iteration, the number *i* of the rule applied is determined and, at the next iteration, the corresponding system R_i is applied.

For example, for the above-mentioned system that simulates the functioning of the above finite-state automaton, the collection of systems is of the form

```
/* R1, R2 */
rs(q,h,t,x)(
        (q1,h,t) = (5,(q2,t)),
isnum(h)–>(
        (q1,h) = (6,1)
),
        (q,x) = (9,0)
);
/* R3, R5, R7 */
rs(q,h,t,x) (
isnum(h)–>
        (q2,h,t) = (7,(q2,t)),
isnum(h)–>(
        (q2,h) = (8,1)
),
        (q,x) = (9,0)
);
/* R4, R6, R8, R9 */
rs(q,h,t,x) ();
```

Next, the procedure appls must first apply the complete system of rewriting rules and then, depending on the number of the last rule applied, it applies the corresponding reduced systems.

Comment 4. Many basic strategies (ntb, nbt, etc.) use appls and, hence, their operation speed will also increase.

The strategy ntb can be similarly optimized but the subterms of the right sides must be unified instead of the right sides.

The possibilities of optimization in systems based on rewriting rules are by no means exhausted by the approach considered in this paper. A deeper optimization can be obtained with the use of the idea of mixed computations, as is done in [10].

REFERENCES

1. J. A. Bergstra, J. Hearing, and P. Klint (eds.), Algebraic Specifications, ACM Press, Addison-Wesley, Reading, Mass. (1989).
2. P. Lincoln, M. Clavel, S. Eker, and J. Meseguer, “Principles of Maude,” in: J. Meseguer (ed.), Proc. 1st Intern. Workshop on Rewriting Logic, Vol. 4, Asilomar, California (1996) (Electronic Notes Theor. Comp. Sci.).
3. H. Kirchner, P.-E. Moreau, P. Borovanskiy, C. Kirchner, and M. Vittek; “A logical framework based on computational systems,” in: J. Meseguer (ed.), Proc. 1st Intern. Workshop on Rewriting Logic, Vol. 4, Asilomar, California (1996) (Electronic Notes Theor. Comp. Sci.).
4. K. Futatsugi and T. Sawada, “Cafe as an extensible specification environment,” in: Proc. of the Kunming Intern. CASE Symp. (Kunming, China) (1994).
5. A. A. Letichevskii and Yu. V. Kapitonova, “Algebraic programming in the APS system,” in: Proc. ISSAC’90 (Tokyo), ACM, New York (1990), pp. 68-75.
6. A. A. Letichevskii, Yu. V. Kapitonova, and S. V. Konozenko, “Optimization of algebraic programs,” in: Proc. ISSAC’91 (Bonn), ACM, New York (1991), pp. 370-376.
7. A. A. Letichevskii, Yu. V. Kapitonova, and S. V. Konozenko, “Computations in APS,” Theor. Comp. Sci., **119**, 145-171 (1993).
8. J.-P. Jouannaud and S. Kaplan (eds.), Proc. Intern. Workshop on Conditional Term Rewriting Systems, Springer, Berlin (1988).
9. P. Lescanne (ed.), Rewriting Techniques and Applications, Lect. Notes Comp. Sci., **256** (1987).
10. A. A. Letichevskii, “Mixed computations and optimization of programs,” Programming, No. 1, 69-76 (1990).