Development of rewriting strategies

A.A. Letichevsky

Glushkov Institute of Cybernetics,
Ukrainian Acad. of Sciences
Kiev 252207, Ukraine
Phone: (044) 266 00 58
E-mail: let%d105.icyb.kiev.ua.@relay.ussr.eu.net

Abstract

Algebraic program is considered as a system of rewriting rules jointly with rewriting strategy that may be represented in procedural form. The optimization of an algebraic program therefore is reduced to the development of optimal strategy and optimal implementation of it. This paper is devoted to the problem of finding optimal strategies for the given set of rewriting rules. New algorithm which implement the strategy of call by need type is proposed for regular systems. This algorithm generalizes previously developed approaches based on strongly necessary occurrences (Huet and Levy) and strongly necessary sets (Sekar and Ramakrishnan). The possibility of extending call by need strategies to a more complex environment of rewriting provided by APS is discussed.

1 Introduction

Rewriting technique is the main technique used in algebraic programming. It has been intensively studied in the ast few years [9, 6]. There are many implementations of term rewriting systems. Some of them support algebraic specifications (ASF [2], ASSPEGIQUE [3]), others are rewriting laboratories based on Knuth-Bendix algorithm for computing canonical systems from a set of equational axioms (REVEUR3 [7], for instance). The languages of OBJ family [4] and O'Donnell's languages [13] are the basis for equational programming. Rewriting technique was used in ANALITIC [16]. It is used in most of modern computer algebra systems from Reduce to Mathematica^{TM} and Axiom.

Rewriting technique is used as a basis for algebraic programming in APS [10, 11] which is under development in Glushkov Institute of Cybernetics of the Ukrainian Academy of Sciences. This system is in some sense the evolution of ANALITIC. It is a professionally oriented instrumental tool for the design of applied systems based on algebraic and logical models of subject domains. APS integrates four main paradigms of programming: procedural, functional, algebraic and logical. This integration is achieved by an adjusted use of corresponding computational mechanisms.

Differently from the traditional approach oriented to the use of canonical systems of rewriting rules with "transparent" strategy of their application, in APS it is possible to combine arbitrary systems of rewriting rules with different strategies of rewriting. Such an approach essentially extends the possibilities of rewriting technique enlarging the flexibility and expressibility of it. There are many examples in which complex algorithms have short and expressive representation in the form of rewriting system with properly chosen rewriting strategy.

The main computational mechanisms of APS include: rewriting strategies, canonical forms, data types, recursive data structures processing, inheritance and interaction between modules. They were formally described in [11]

The hierarchical structure of the system provides different tools for increasing the efficiency of algebraic programs. It includes writing canonical forms for most frequently used operations at low level (C language), writing internal procedures in C, sharing the complexity between rewriting rules and strategies, developing efficient strategies and implementing them.

When a good strategy is found it may be optimized using formal transformations of procedural programs. A corresponding approach based on mixed computations was described by in [12]. Nearby results were obtained in [15]. This paper is devoted to the problem of developing optimal strategies for a given set of rewriting rules.

One of the first studies of optimal strategies for *regular* rewriting systems has been done by Huet and Levi in [5]. In that paper the notion of *needed redex occurrences* was introduced and the strategy that reduces only needed occurrences was developed for a class of regular rewriting systems called *strongly sequential*. The notion of strong sequentiality as well as the strategy based on this notion depends only on the set of left hand sides (lhss) of rewriting systems. Further development and study of the notion of strong sequentiality was presented in [8].

Recently some new generalizations for call by need have been developed. In [1] the notion of tree sequentiality was introduced that is more general than strong sequentiality. In the paper of Sekar and Ramakrishnan [14] a nice generalization of Huet-Levy theory was proposed based on the notion of *strongly necessary sets* of occurrences and the algorithm was developed that finds minimal (in some sense) strongly necessary sets and uses them for optimal reduction. In the special case of strongly sequential systems this algorithm finds one of the needed occurrences and realizes Huet-Levy strategy.

In this paper new call by need strategy will be described. This strategy is based on the notion of necessary sets of redexes (occurrences) like Sekar-Ramakrishnan algorithm but removes some essential restrictions for it. For example it does not demand the rewriting system to be constructor and remains optimal even if the source term contains occurrences that are not legal. It is shown also how to implement the optimal strategy in APS.

In the next section the tools for building strategies used in APS are described. Then the criteria of optimality will be discussed. The call by need strategy and its implementation in APS at the high level is considered in section 4.

2 Tools for building strategies

Example of algebraic program. Let us begin with a typical example of algebraic program written in the APLAN language for APS. This program is called `log.ap` and contains some tools to process propositional formulas.

```
INCLUDE <ac.ap>
MARK subs(2);
NAMES R,R1,Q1,cnf;
/*         Rules for eliminating <=>, ->, and de Morgan rules        */
R :=rs(x,y)(
         x <=>  y = (x -> y) & (y -> x),
         x  ->  y = ~(x) || y,
       ~( ~(x)  ) = x,
       ~(x ||  y) = (~(x) & ~(y)),
       ~(x  &  y) = (~(x) || ~(y)),
       ~(x <=> y) = (~(x -> y) || ~(y -> x)),
       ~(x  -> y) = (x & ~(y))
);
/*                          Rules for CNF                             */
R1 :=rs(x,y,z,u,v)(
       (x & y || z & u) & v = (x || u) & (y || z) & (y || u) &(x || z) & v,
       (x & y || z) & u = ( y || z) & (x ||z) & u,
       (x || y & z) & u = ( x || z) & (x || y)  & u,
       x & y ||  z  & u = ((x || z) & (y || z)) & (x || u) &(y || u) ,
       x & y ||  z      = ( x || z) & (y || z),
       x || y &  z      = ( x || y) & (x || z)
```

```
);
Q1 :=rs(x,y,z)(
       (x  & y)|| z  = (x || z) & (y || z) ,
        x ||(y  & z) = (x || y) & (x || z),
       (x || y)|| z  =  x || y  || z
);
cnf:=proc(x)(
       ntb(x,R),
       can_ord(x,R1,Q1),
       return(x)
);
NAMES deM,Id,ntb2,is_id;
deM :=rs(x,y)(
       ~( ~(x)  ) = x,
       ~(x ||  y) = deM(~(x)) & deM( ~(y)),
       ~(x  &  y) = deM(~(x)) || deM(~(y))
);
/*           Rules for proving identities in logic                    */
Id:=rs(x,y,z)(
        1       ->      0  = 0,
        0       ->      x  = 1,
        x       ->      x  = 1,
        x       ->      1  = 1,
        x       -> y || z  = x & deM( ~(y) ) -> z,
        x       -> y &  z  =(x -> y) &( x -> z),
        x || y  ->      z  =(x -> z) &(  y -> z),
        x &  y  ->    ~(z) = subs(z=1,x)  -> deM( ~(subs(z=1,y))),
        x &  y  ->      z  = subs(z=0,x)  -> deM( ~(subs(z=0,y))),
        x       ->      y  = 0
);
ntb2:=proc(t,R)(
          appls(t,R);
          (ART(t)>0)->ntb2 (arg(t,1),R);
          t:=can(t);
          ntb2 (t,R)
);
is_id:=proc(x)(
        ntb(x,R);
        x-->(1->x);
        ntb2(x,Id);
        return(x)
);
```

The first line of `log.ap` means that it includes previously defined algebraic program `ac.ap` which contains some standard definitions and the description of associative-commutative operations. In particular it contains the description of logical connectives `~`(negation), `&`, `||` (disjunction), `->`, `<=>`, and information that defines `&` and `||` as boolean operations. The next line introduces new operation symbol `subs` with arity 2. To evaluate algebraic programs different interpreters may be used. The interpreter `nsint` which is to be used for evaluating `log.ap` interprets `subs` as the substitution function: `subs`$((x_1 = y_1, \ldots, x_n = y_n), z)$ substitutes $y_1, \ldots, y_n$ for all occurrences $x_1, \ldots, x_n$ in z where $x_1, \ldots, x_n$ are supposed to be different atoms. At the time of substituting the expression is simplified by reducing it to the main canonical form which will be discussed below.

The values of the names `R`, `R1`, `Q1`, `deM` and `Id` defined by initial assignments are systems of rewriting rules. To apply them to algebraic expressions (terms) one may use standard strategies implemented by the current interpreter or write his own ones. The function `cnf` computes (some) conjunctive normal form of logical expression. It is defined by means of procedure that uses two standard strategies `ntb` and `can_ord`. The first is a one time top-bottom strategy. It passes over the nodes of a tree represented expression in a top-bottom manner and checks the applicability of rewriting rules in the order they are written in the system. This strategy is used for eliminating `->` and `<=>` and transferring negations by de Morgan rules.

Strategy `can_ord` works with two systems of rewriting rules. The first system is applied top-bottom, the second – bottom-up. When the strategy passes over the nodes bottom-up the subterms are ordered w.r.t. ac- and boolean operations by merging already ordered arguments of such operations. The laws of contradiction and exclusion third are also used when merging is applied for boolean operations. It is important to note that after each successive application of a rule the substituted instance of the right hand side (rhs) is reduced to its *main canonical form*. This reduction varies from one interpreter to another and usually includes constant computations for arithmetical and logical operations, computations for interpreted operations (such as `subs`) and some other simplifications.

The system `Id` is used for checking the logical formulas to be identically true. If so the formula is transformed to 1, otherwise to 0. The system `Id` is not confluent but the result will be defined uniquely if the strategy meets two conditions: it checks the rules in the order they are written and the rewriting terminates only when no rules are applicable. The standard strategy `applytb` which repeats `ntb` while possible would be sufficient but it is too slow because while reducing a formula $X\&Y$ it will reduce Y even if X is already reduced to 0. The user defined strategy `ntb2` is much faster. It uses the function `can` which calls main canonical form reduction that in particular applies identity $0\&X = 0$. The statement `appls(t,R)` applies the system `R` to the top operation of `t` while possible.

Basic strategies. The strategies of rewriting in APS are based on two main internal (that is realized at low level) procedures `applr` and `appls`. The statement `applr(`t`,`R`)` attempts to apply one of the rules of the system R to the term t. If there are no applicable rules, the name `yes` gets the value 0. Otherwise, the first applicable rule is applied and `yes` gets the value 1. The application of unconditional rule is usual: matching left hand side (lhs) with t, if success then substitution of the values of variables in the rhs and replacement of t. Before replacing the redex, the instance of the rhs is reduced to the main canonical form.

Conditional rules are applied to terms in the following manner. First matching is to be done. Then if success the condition is reduced to main canonical form. If the result is 1, applying the rule continues as usual. Otherwise application is cancelled.

Besides the condition "yes" different versions of the procedure `applr` may produce some additional information that may be used by strategies for selection of further movement of rewriting process. For example, it may be the number of rule being applied, the right hand side of this rule and so on. The statement `appls(`t`,`R`)` calls `applr(`t`,`R`)` while `yes` $= 1$.

There are some peculiarities of rewriting in APS that must be pointed out. The first is strict order of testing rules for applicability. It means that next rule is applied only if all previous rules are not applicable. If no one of the lhss of the system is an instance of another, this order is not essential because there may be only one applicable rule for any given term (or given occurrence). Any system may be transformed to such *order independent* system in the following way. Get some lhs and substitute for the variables all possible most general terms such that the new lhs and the others are not the instances of one another and make the corresponding replacements. For a finite signature there may be only a finite number of changes. For example, the lhs of the last rule in Id is more general than all others and to make the system be order independent it must be replaced by `1->~(x)=0, ~(x)->0=0, ~(x)->1=0, ~(x)->a_1=0,...,~(x)->a_n=0,` where $\mathtt{a}_1, \ldots, \mathtt{a}_n$ are all propositional variables that may occur in the expression. Therefore a rewriting system may be considered as a short form of an order independent system.

Another peculiarity of rewriting in APS is the use of canonical form reductions which are applied to the rhs after each successful rewriting. The *main canonical form* is defined by means of interpreters

of operations and is computed at the time of substituting the rhs by means of the formula

$$\begin{gathered}\mathtt{CAN}(\omega(t_1,\ldots,t_n)) = \varphi_\omega(\mathtt{CAN}(t_1),\ldots,\mathtt{CAN}(t_n));\\ \mathtt{CAN}(x_i) = u_i;\\ \mathtt{CAN}(z) = z;\end{gathered}$$

where $x_1,\ldots,x_n$ are the variables of rewriting system, $u_1,\ldots,u_n$ are their values, computed during the matching, z is primary object (number, name, atom or string). The function φ_ω is the interpreter of the operation ω and is realized as a low level function. For uninterpreted operations it acts as the identity function. There are two important exceptions to these rules: quote operation and application. The following rules are used for them:

$$\begin{gathered}\mathtt{CAN}(`(t)) = t[x_1 \leftarrow u_1,\ldots,x_n \leftarrow u_n];\\ \mathtt{isfun}(f) \Rightarrow \mathtt{CAN}(f(x)) = \varphi_f(x).\end{gathered}$$

$\mathtt{isfun}(f)$ is true if f is the name of a function, that is the value of this name is a rewriting system or a procedure that returns a value. $\varphi_f(x)$ is the value of the corresponding function on x. Apart from that if f is the name of a rewriting system, the value is computed by applying this system to x by means of `applr`. The semantics of application allows to introduce canonical reductions at high level as interpreted functions. The function deM from the program `log.ap` may serve as an example. The quote operation helps to switch off canonical reduction if it is necessary.

Some system interpreters of APS implement more complicated mechanisms for computing canonical forms that support efficient computations in multisorted algebras with polymorphic operations (see [11]).

APS permits also the use of ac-operations (associative and commutative ones). There are two kinds of ac-operations: arithmetic and boolean type. An arithmetic ac-operation is considered as an operation of a free module over integers, a boolean type operation is idempotent and satysfies identities connected with negation. Canonical forms for ac-operations include ordering, reduction of similars for arithmetical operations, applying the idempotency law, the laws of contradiction and exclusion third for boolean type ones. It also includes the main canonical form reduction. It would be inefficient to use ac-canonization at the level of `CAN` computation. Therefore in APS ac-canonization is used explicitly at the level of strategies. The strategy `can_ord` may be mentioned as an example. It may be represented by means of the following APLAN procedure.

```
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)
);
```

All identities of ac-operations are used in the internal procedure `merge` that reduces a term to its canonical form if this term is represented as ac-operation applied to already canonized arguments.

The main canonical form and ac-operations provide the consideration of algebraic expressions up to some congruence consistent with the identities of the algebra that defines the subject domain. Two expressions are equivalent if they are reduced to the same canonical form (the conditions for operation interpreters to define the canonical form are formulated in [11]).

A strategy is called to be *normalizing* if the result of its application (if defined) is *normalized* (i.e. contains no redexes) and presented in the main canonical form. A strategy is *complete* if it terminates for any term that may be normalized by means of some strategy at all. It may be shown that all strategies used in the example considered above are normalizing and complete w.r.t. the rewriting systems they are applied to. But the uniqueness of the result may be garanteed only for proving identities and eliminating connectives.

Yet another peculiarity of APS is the possibility of identifying the nodes of the graph that represents the term. In particular if the rewriting system is not right linear (some variables may repeat in the rhss) the values substituted for the different occurrences of the same variable are identified. To avoid identification the functions `copy` or `new` may be used.

3 Criteria for optimization

The main efficiency criterion for an algebraic program is the time it spends for normalizing terms. This criterion depends on many factors connected with implementation, therefore inorder to characterize the algorithm realized by the program other criteria must be used. They are the number of attempts to apply the rewriting system, the number of successful attempts (reductions), their ratio. More detailed criteria include the number of nodes observed during the execution of the program, the number of comparisons and other actions that demand the time to be spent for.

Call by need strategies introduced by Huet and Levi are directed to decrease the number of reductions by choosing only needed redex occurrences. The redex occurrence is called *needed* if this occurrence or one of its residuals (copies of a subterm that appear as a result of rewriting) must be rewritten in arbitrary possible rewriting processes. Call by need strategies are developed for regular systems (left linear and nonoverlapping). For such systems any redex occurrence which is not rewritten at a current step of rewriting transforms to an unchanged copy and therefore left to be redex occurrence if it does not disappear at all. Needed redexes may not disappear and the strategy which rewrites only needed redexes avoids wasteful rewriting redexes that will disappear, therefore it decreases the total number of reductions.

The notion of needed redexes is unsolvable and Huet and Levy restricted their consideration to *strongly needed redexes* (their needness may be found out considering only the lhss of the system) and the subclass of strongly sequential systems for which a call by need strategy may be effectively constructed. The generalization proposed by Sekar and Ramakrishnan extends the notion of needness to the sets of redexes. The set of redex occurrences is called *necessary* if at least one of them or their residuals must be reduced in any reduction process. And again for *strongly* necessary sets and regular constructor systems the algorithm is developed that finds some necessary set which is proved to be minimal for legal terms.

The search for necessary occurrences needs a large amount of work which may appear to be vain if every variable of lhs occurs in corresponding rhs (redexes never disappear). But information obtained during search of redexes may be used to improve the rewriting process w.r.t. other criteria: to decrease the number of matchings, number of nodes being observed and so on. This is true for the new strategy considered in the next section. Besides it is free of the restrictions of Sekar-Ramakrishnan method.

4 Call by need strategies

Regular rewriting systems are of great importance in the theory and applications of rewriting technique.

They are defined as *leftlinear* (each variable in the lhs of each rule occurs only once) and *nonoverlapping* (no left hand side is unified with a subterm of another, or there is no critical pairs). Regular systems are confluent but not necessarily noetherian.

The procedure `nset` presented below is based on the notion of strongly necessary sets and the modification of `applr`.

The modified procedure `applr(t,R)` does the same as the original one but in addition to producing the value of the name `yes` it produces as the value of the name `failset` the set of occurrences which in the case when `yes=0` satisfy the *completeness* condition w.r.t. `t` and the set L of lhss of the system `R`. To formulate this condition let us introduce the notion of *compatibility* of the term t with the lhs l from the set L of lhss. This notion is recursive: t is *compatible* with l if it is an instance of l or there exist nonempty disjoint occurrences $p_1, \ldots, p_n$ such that subterms $\mathtt{arg}(t,p_1), \ldots, \mathtt{arg}(t,p_n)$ corresponding to these occurtences are compatible with some lhss from L and $t[p_1 \leftarrow t_1, \ldots, p_n \leftarrow t_n]$ is the instance of l for some $t_1, \ldots, t_n$ (denotation $\mathtt{arg}(t,p)$ is used instead of usual t/p because it is accepted for APLAN).

Suppose that t is not an instance of any lhs from R. The set $\{p_1, \ldots, p_k\}$ is *complete* w.r.t. t and L if there exists a subset $\{l_1, \ldots, l_k\}$ of the set L such that $\mathtt{arg}(t,p_i)$ is not an instance of $\mathtt{arg}(l_i,p_i)$, $i = 1, \ldots, k$, t is compatible with no lhs from $L \setminus \{l_1, \ldots, l_k\}$ and for each $() < q < p_i$ $\mathtt{arg}(t,q)$ is not compatible with any lhs from L, $i = 1, \ldots, k$. Note that if $k = 0$ (the set of occurrences is empty) then completeness means that t is compatible with no lhs from L.

By definition [14] the set Q of redexes is strongly necessary w.r.t. L if in an arbitrary reduction sequence by means of an arbitrary rewriting system with the set L of lhss at least one of the redexes in Q or its residual is reduced on some step of rewriting.

Theorem 1 *Let $\{p_1, \ldots, p_k\}$ be complete w.r.t. t and L, the sets $Q_1, \ldots, Q_k$ are strongly necessary for $\mathtt{arg}(t,p_1), \ldots, \mathtt{arg}(t,p_k)$ correspondingly. Then if $Q_1 \cup \ldots \cup Q_k$ is not empty, this union is a strongly necessary set for t, otherwise a nonempty strongly necessary set for any of $\mathtt{arg}(t,i)$ is strongly necessary for t.*

The term t can not be reduced at the root before reducing one of the subterms $\mathtt{arg}(t,p_1), \ldots$ $\ldots, \mathtt{arg}(t,p_k)$. But these terms can not be reduced before at least one from the union $Q_1 \cup \ldots \cup Q_k$. And if this union is empty t can not be reduced at all and any strongly necessary set for its arguments is strongly necessary for t.

To be effective the modified `applr` must use some simple sufficient condition for noncompatibility which might be checked simultaneously with matching. Such simple conditions exist for so called constructor systems that distinguish between defined and constructor operations: a term with constructor operation at the root may never be compatible with any lhs. Exactly this kind of systems is considered in [14] and our algorithm generalizes their approach to nonconstructor systems.

The strategy `nset` based on strongly necessary sets and theorem 1 may now be represented as follows.

```
nset:=proc(t,R)loc(cont,s,i)(
      dowhile(
           cont:=applns(t,R),
      cont);
      forall(s=arg(t,i),
           nset(s,R)
      )
);
applns:=proc(t,R)loc(cont,fs,p)(
     applr(t,R);
     yes->return(1);
     cont:=0;
     fs:=failset;
     nonempty(fs)->
```

```
          forall(p in fs,
              cont:=cont||applns(arg(t,p),R)
          );
     return(cont)
);
```

The loop `forall(s=arg(x,i),y)` means the same as

```
for(i:=1,i<=ART(x),i:=i+1, s-->arg(x,i),y)
```

The loop `forall(x in y, z)` executes `z` for all elements of the list `y`.

Every reduction that is made by the algorithm in one step, that is in the outermost call of `applns` rewrites only redexes that belong to some strongly necessary set. It may be shown that this set is included into the set, generated by Sekar-Ramakrishnan algorithm, and therefore for strongly sequential systems it consists with the unique strongly necessary occurrence.

Improvements of nset. There are some possibilities to improve the above algorithm. First the necessary set which is computed after defining the failset may be reduced dynamically during computation. Indeed, if in the loop for all p in failset applns has rewritten the top occurrence of arg(t,p) the term t may become redex and then it is not necessary to continue the search for other elements of low level necessary set. The second improvement is the decrease of the number of nodes being observed in the process of rewriting. This may be achieved by combining the search for necessary sets with the process of rewriting. The improved algorithm may be represented in the following way.

```
nset:=proc(t,R,L)loc(cont,s,i)(
      dowhile(
           cont:=applns(t,R,L),
      cont>0);
      forall(s=arg(t,i),
           nset(s,R,L)
      )
);


applns:=proc(t,R,L)loc(cont,cont1,l,q)(
     cont:=0;
     forall(l in L,
          is_type(t,l)->(
               q-->compat(t,l);
               equ(q,match)->(
                    applr(t,R);
                    return(2)
               )else q-->arg(q,1);
               cont1:=applns(q,R,L);
               (cont1==2)->(
                    applr(t,R);
                    yes->return(2)
                    else cont1:=1
               );
               cont:=cont||cont1
          )
     );
     return(cont)
);
```

```
compat:=proc(t,l)loc(p,q,i)(
     is_par(l)->return(match);
     p:=match;
     is_type(t,l)->(
          for(i:=1,i<=ART(t),i:=i+1,
               q-->compat(arg(t,i),arg(l,i));
               equ(p,match)->p-->q
          );
          return(p)
     );
     return(pt(t))
);
```

The procedure `compat` returns atom `match` if `t` is matched with l or the pointer (operation `pt`) to the first subterm of `t` which is not matched with the corresponding subterm of `l`. The procedure `applns` returns now 0,1 or 2. It returns 0 if the term `t` can never be reduced at the root. The value 1 means that some necessary set of occurrences in `t` was reduced but `t` can not be reduced at this moment. The value 2 means the same but it is possible for `t` to be rewritten.

The proof of correctness of the program `applns` is realized using induction on the depth of the term and the following invariant for the main loop. If t is the initial value of `t` and $l_1, \ldots, l_n$ are the lhss already observed in the loop `forall(l in L,...)` then there exists the set of occurrences of t which is complete w.r.t. t and $l_1, \ldots, l_n$ and the union of some necessary sets for these occurrences has been already reduced.

The procedures `nset` and `applns` allow some other improvements which eliminate repeated actions such as repeated observing of subterms which are compatible with no lhss, but the main optimization may be obtained by means of mixed computations by substituting diferent rewriting systems into the strategy [12].

The strategy `nset` may also be applied to nonregular systems and after some modifications to the systems with APS semantics (ordering, canonical forms and identification of nodes). But it may lose the normalizing property and completeness. The repetition of the strategy while possible makes it normalizing but completeness requires special investigations. In practice these problems are not very difficult and the strategy may be effectively used for the extended classes of the systems.

5 Concluding remarks

The strategy based on the necessary sets of occurrences which was described in the paper was realized in APS and tested on some examples from computer algebra and logic. The experiments have shown that this strategy may be more efficient than others in spite of the time spent for analysis of terms. Therefore this strategy has not only theoretical but also practical interest and is worth studying in more details.

References

[1] V. Ambriola and A. Salibra. Tree sequential term rewriting system. TR-40/89, University of Pisa, 1989.

[2] J. A. Bergstra, J. Hearing, and P. Klint, editors. *Algebraic Specification.* ACM Press and Addison-Wesley, 1989.

[3] M. Bidoit and C. Choppy. ASSPEGIQUE: an integrated environment for algebraic specifications. In *Proc. Intern. Joint Conf. on Theory and Practice of Software Development*, pages 246–260. Springer-Verlag, 1985.

[4] J. Goguen, C. Kirchner, H. Kirchner, A. Megrelis, and T. Wincler. An introduction to OBJ-3. In Jouannaud and Kaplan [6].

[5] G. Huet and J. J. Levy. Computations in nonambiguous linear term rewriting systems. Technical Report 359, INRIA, Le Chesnay, France, 1979.

[6] J.-P. Jouannaud and S. Kaplan, editors. *Proc. 1st Intern. Workshope on Conditional Term Rewriting Systems*. Springer-Verlag, 1988.

[7] C. Kirchner and H. Kirchner. Reveur3: Implementation of a general completion procedure parametrized by built-in theories and strategies. *Science of Computer Programming*, 20(8):69–86, 1986.

[8] J. W. Klop and A. Middeldorp. Strongly sequential term rewriting system. TR CS-R8730, Centre for Mathematics and Computer science, Amsterdam, 1988.

[9] P. Lescanne, editor. *Rewriting Techniques and Applications*, volume 256 of *LNCS*. Springer-Verlag, 1987.

[10] A. A. Letichevsky and J. V. Kapitonova. Algebraic programming in APS system. In *Proc. of ISSAC '90*, pages 68–75, Tokyo, Japan, August 20–24 1990. ACM, New York.

[11] A. A. Letichevsky, J. V. Kapitonova, and S. V. Konozenko. Computations in APS. In *Actes preliminaires, du symposium Franco-Sovietique Informatika 91*, pages 133–153, Le Chesnay, France, 1991. INRIA.

[12] A. A. Letichevsky, J. V. Kapitonova, and S. V. Konozenko. Optimization of algebraic programs. In *Proc. of ISSAC '91*, pages 370–376. ACM Press, 1991.

[13] M. J. O'Donnell. Term rewriting implementation of equational logic programming. In Lescanne [9], pages 1–12.

[14] R. C. Sekar and I. V. Ramakrishnan. Programming in equational logic: Beyond strong sequentiality. In *Proc. 5th Annual IEEE Symp. on Logic in Computer Sci*, pages 230–241. IEEE Comp. Soc. Press, June 1990.

[15] D. Sherman, R. Strandh, and I. Durand. Optimization of equational programs using partial evaluation. In *Proc. of the Symposium on partial evaluation and semantic-based program manipulation, Sigplan Notices*, volume 26, pages 72–82. ACM Press, 1991.

[16] A. A. Stogny and T. A. Grinchenko. Mir series computers and ways of increasing the level of machine intelligence. *Cybernetics (Translated from Russian)*, 23(6):807–817, 1987.