Algebraic Programs Optimization

A.A. Letichevsky, J.V. Kapitonova,
S.V. Konozenko

Glushkov Institute of Cybernetics,
Ukrainian Acad. of Sciences
Kiev 252207, USSR

Abstract

Algebraic program is a system of relations (equalities of data algebra) with a given strategy for applying these relations as rewriting rules. Algebraic program may be optimized by transforming a system of relations or by transforming a strategy. Only second case of optimization is considered in the paper.

The problem of algebraic program optimization is investigated in the context of programming in the APS-1 system. Strategies are represented as procedures, written in APLAN language, and satisfy the locality property. The first goal of strategy optimization is to minimize the number of pattern matching being done in the process of rewriting. When the strategy for a given system of relations is chosen, further optimization may be achieved by transforming algebraic program into procedural one with the goal to decrease the time spent for each pattern matching. At this stage the partial evaluation technique is used. The residual program obtained after substituting relations into a strategy is optimized by ordinary simplifications based on searching the invariant relations and optimal loops unfolding.

1 Introduction

In the paper algebraic program is considered as a set of relations (equalities) of data algebra with the strategy for applying these relations as rewriting rules to algebraic data structures in order to obtain the result. Data structures are objects that represent expressions (terms) of the data algebra. Usually they are trees but acyclic graphs or even graphs with cycles are also admissible.

The purpose of program optimization is to reduce computation time and it may be achieved by two possible ways. The first way is to decrease the number of pattern matching being done in the process of rewriting by choosing a suitable computational strategy. The second way is to decrease the total matching time by means of optimizing transformations of chosen computational strategy with taking into account the structure of a system of relations.

The implementation of the second way of optimization naturally leads to the use of the idea of partial computations [1]. Indeed, the strategy `apply(`t`,`R`)` is a program for computing the function of two arguments: the expression t of data algebra and the system of relations R. Making the system of relations R to be fixed, one may compute the residual program `apply_R(`t`)` and its optimization gives the required result. To realize optimization of the residual program one may use search of invariants and loops unfolding. The basic ideas of such optimization are considered in [5].

Programming by rewriting rules as a special kind of programming (equational programming) was convincingly presented by O'Donnell in [9]. There are research works dedicated to problems of effective pattern matching [10] and algebraic program optimization [2, 11]. However the canonical conditions for systems of rewriting rules are essential in these works. Research work described in the current paper is performed in the context of algebraic programming system APS-1 which was developed in the Institute of Cybernetics of the Ukrainian Academy of Sciences. General description of the system was published in [8, 6]. More details may be found in [7]. Decomposition of an algebraic program into a system of relations and a computational strategy allows to give up the canonical restriction and to control computations flexibly by combining systems of relations with various computational strategies. Automatic synthesis of optimal algebraic programs allows to extend considerably abilities of using algebraic programming for solution of complex applied problems.

2 Computational strategies

Computational strategies in the system APS-1 are determined by procedures including basic procedures `applr(`t`,`R`)` and `appls(`t`,`R`)`. The statement `applr(`t`,`R`)` attempts to apply once the relations of the system R to the expression t. If there are no applicable relations, the name `yes` gets the value 0. Otherwise, the first applicable relation is applied and `yes` gets the value 1. The statement `appls(`t`,`R`)` calls `applr(`t`,`R`)` while `yes` $= 1$.

The procedure `applr` besides of the condition `yes` produces the number of the relation that had been applied to the term t the last time as a value of the name `rnum` (if R was not applied at all, `rnum` $= 0$). It allows to select the continuation of the strategy execution more subtly.

The implementation of the `applr` in the system APS-1 has some peculiarities, which will be ignored in this consideration. Besides, by practical reasons only local strategies represented in the procedural subset of APLAN language will be considered in the paper.

The strategy of one time application of the system R from top to bottom may be described in APLAN by the following procedure:

```
ntb:=proc(t,R)loc(i)(
  appls(t,R);
  for (i:=1, i<=ART(t), i:=i+1,
    ntb(arg(t,i),R)
  )
);
```

The bottom-up strategy differs from the top-bottom one time so that the statement `appls(t,R)` follows the loop over the arguments of the expression `t` in its procedural description. The following strategy determines two times going around of an expression: at first top-bottom and then bottom-up. When going back after successful application, top-bottom going is repeated until the first unsuccessful matching. By doing so the system R is decomposed into two subsystems R_1 and R_2. The first one is applied when going forward, the second one when going back. As the described strategy has analogy to attribute grammars, it is called `apply_atr`.

```
apply_atr:= proc(t,R1,R2)loc(i)(
  appls (t,R1);
  for (i:=1, i<=ART(t), i:=i+1,
    apply_atr (arg(t,i),R1,R2)
  );
  apply_next(t,R2)
);
apply_next:= proc(t,R)loc(i)(
  appls (t,R),
  yes->
    for (i:=1, i<=ART(t), i:=i+1,
      apply_next (arg(t,i),R)
    )
);
```

The `apply_atr` strategy is near to optimal one, because it provides no more than two time observation of any node of `t` if it is represented by the tree.

Let us consider the problem of finding the optimal strategy in the case of the canonical (neuterian and confluent) system of relations R. In this case one can use any complete strategy (which stops only if the system R is applicable to no one of subexpressions of the expression t). Multi-time top-bottom or bottom-up strategy may be used as complete one. The strategy `apply_atr(`t`,`R`,`R`)` generally speaking is not complete but it may be transformed to complete one if the call of `apply_next` in the loop for `i` in the description of `apply_next` will be changed to the call of `apply_atr`. At the same time new strategy might considerably increase the number of repeating observations of the nodes of t.

To find the optimal strategy for canonical systems of relations it is necessary to analyze dependencies between their left and right sides. Let us define partial order on the set of parameterized expressions (expressions with indicated occurrences of parameters used for pattern matching). Assume that $t \le s$, if t can be unified with some subexpressions of the expression s, the parameters of s having been renamed in order to differ from the parameters of t.

Let $R = (l_1 = r_1, \ldots, l_m = r_m)$ be a system of relations. Denote $dn(t) = \{i = 1, \ldots, m \mid l_i \le t\}$, $up(t) = \{i = 1, \ldots, m \mid t \le l_i\}$. Evidently, when the strategy moves top-bottom and the relation $l_i = r_i$ such that $up(r_i) = \emptyset$, has been applied, then moving to the top will not provide successful application of the system R. The reason is that there is no a node, to which the system R is applicable, on the trace from the root node of the expression t to the current node. Analogously, when going bottom-up one should not return down if $dn(r_i) = \emptyset$. It follows from this, that if $dn(r_i) = \emptyset$ for all $i = 1, \ldots, m$, then the best strategy for R is one time bottom-up application, and if $up(\mathtt{r}_i) = \emptyset$ for all $i = 1, \ldots, m$, then the best strategy is one time top-bottom.

In general case the modified `apply_atr` strategy which is called `apply_atr_1` may be used. It has only two parameters `t` and `R` and instead of `apply_next` includes more complex procedure `apply_next_1`. This procedure prescribes after successful application to return down not for one step but for the determined number of steps depending on the last applied relation.

```
apply_next_1:= proc(t,R,n)loc(i,j)(
  n>0->
    for (i:=1, i<=ART(t), i:=i+1,
      apply_next_1(arg(t,i),R,n-1)
    );
```

```
    applr (t,R); j:=rnum;
    j>0->
      for (i:=1, i<=ART(t), i:=i+1,
        apply_next_1(arg(t,i),RN(j),dn(j)
      )
);
```

This procedure is called from `apply_atr_1` by the statement `apply_next_1(t,R,0)`. The function `dn`(j) computes the maximal depth at which in the expression r_j may occur the subexpressions, which the relations from R may be applicable to. The set `RN`$(j) \subset R$ consists of the relations, left sides of which are unified with the subexpressions of the expression r_j.

The resulting program `apply_atr_1`(t, R) is almost optimal. Indeed, assume all right sides of the relations from R to be represented by balanced trees. Then each node of the tree representing t, as well as any new node appeared in the rewriting process are observed by the strategy not more than twice. In the case of nonbalanced trees the number of observations may increase, but it still remains limited. In order to overcome the nonbalance effect it is possible to introduce auxiliary data structures, which for each expression r_j indicate exact locations of all subexpressions, unified with left hand sides of the relations.

The technique described here can be used not only for canonical relation systems but also for algebraic programs which combine arbitrary relations with proper strategy. It works also in combination with special peculiarities of APS-1 such as basic canonical forms, use of arbitrary graphs instead of trees and so on.

3 Mixed computations over strategies

Suppose that optimal or near to optimal strategy w.r.t. the number of matchings is chosen. The purpose of further optimization is decreasing the total time spent for matching processes.

It is well known that the working time of `applr` procedure may be done dependant of maximal complexity of one relation only but not of the number of relations. In APS-1 system this effect is achieved by means of generating special data structure that helps to realize fast matching. Such optimization however does not take into account that the time for the next step matching may be shortened by use of information obtained on the previous steps. This information may be accumulated and supported in the proper data structures or in the program states. The last though increases the size of program but provides more fast matching because does not demand auxiliary time for additional computations. The general plan of optimizing transformations of algebraic program with given relation system R and strategy `apply` may be described by the following way.

Let us consider the statement `apply`(t, R). Suppose the system $R = (l_1 = r_1, \ldots, l_m = r_m)$ to be fixed and synthesize the procedure (residual program) `apply_R`(t) with one parameter t such that statements `apply`(t, R) and `apply_R`(t) are equivalent.

The first step constitutes in substitution of the system R into the body of procedural specification of `apply`. Supposing that `apply` uses the calls of the procedures `applr` and `appls` it is necessary to change these calls for the bodies of their procedural descriptions. It is best to take the simplest versions of their implementation with complete looking through all of the relations. For instance:

```
appls:= proc (t,R)(
  yes:= 1;
  while (yes, applr(t,R))
);
applr:= proc (t,R) (
  for all (l=r) in R(
    init(v);
    match(t,l);
    yes -> t:=subst(r,v)
  )
);
```

Here `v`(u) is the value of the parameter u of the system R, `init(v)` makes all of the values of `v`(u) equal to `nil`, `subst` substitutes the values of parameters found as a result of matching into the right hand part of the relation $l = r$. The substitution of R makes it possible to unfold the loop to the sequence of statements:

```
init(v);
match (t,l1);
yes -> t:=subst(r1,v);
init(v);
match(t,l2);
...........
init(v);
match(t,lm);
yes -> t:=subst(rm,v);
```

The matching procedure is recursive. It reduces the matching `match(t,`l`)` to the matching `match(arg(t,`i`),arg(`l`,`i`))` and comparison `compare(t,v(`u`))` of the expression `t` with the previously obtained value `v`(u) of the parameter u if $l = u$. For concrete patterns $(l_1, \ldots, l_m)$ the recursion is completely eliminated and the body of the procedure `applr` is reduced to the sequence of comparisons of the types of expressions: two expressions have the same type if their main operations are the same.

Consider an example. Let

```
R:=rs(a,b,c,d)(
   (a||b)&c=>d)  =   (a&c=>d)&(b&c=>d),
       (a&b=>d)  =   ((a,b)=>d)
);
```

After the substitution of R, unfolding of the loop over relations, recursion elimination, simplification of conditional statements, systematical introducing auxiliary pointers for the nodes of the type $\texttt{arg}(x, i_1, \ldots, i_k)$ and some other transformations the body of applr may be transformed to:

```
(type (t)!=(()=>())) -> goto N;
x1 --> arg(t,1);
type(x1)!=(()&()) -> goto N;
x11--> arg(x1,1);
type(x11)!=(()||()) -> (
  v(a) --> x11;
  v(b) --> arg (x1,2);
  v(c) --> arg (t,2);
  t:= ((v(a),v(b)) => v(c) );
  goto M;
);
v(a) --> arg(x11,1);
v(b) --> arg(x11,2);
v(c) --> arg(x1,2);
v(d) --> arg(t,2);
t:=(v(a)&v(c)=>v(d))&(v(b)&v(c)=>v(d));
goto M;
```

Note that the names in the APLAN procedures are used as pointers. The statement x-->y means the setting of the pointer x to the main node of y and x:=y replacing of the main node of object pointed to by x by the copy of the main node of the object pointed to by y.

The labels M and N correspond two different exits from applr — the case of successful and unsuccessful application of the system. Corresponding goto statements are convenient to use instead of assignments yes:=0 and yes:=1. The first stage of algebraic program apply(t,R) optimization is finished by the construction of optimized residual program Q for the body of applr procedure. At this stage the working time for applr already becomes to be independent of the number of relations. Further optimization is much more deep. It represent the global optimization of strategy by means of the method called as optimal loops unfolding.

4 Generalized loops unfolding

To explain the essence of the method it is necessary to use some formal model of the program such as interpreted program schemata. The notion of U-Y-schema [3] which is similar to transition system is used for this purpose.

Let U be the set of elementary conditions and Y — the set of primary statements (for instance, the set of conditions and statements of APLAN or C language). U-Y-schema $S = (A, T)$ is defined by the set of states A and the set of transitions T. Each transition is the 4-tuple $a \xrightarrow{u/y} a'$, where $a, a' \in A$, $u \in U$, $y \in Y$. In the set A one initial and one or more terminal states are pointed out. Conditions and statements are acted on the set of memory states and transitions have natural interpretation: if $u = 1$, then the statement y is executed and schema comes from the state a to the state a'. Schema may be represented by transition diagrams as well as by texts of the proper procedural language.

Condition u, defined on the set of memory states, is called to be an invariant of the state $a \in A$ of the program S, if $u = 1$ every time when the program comes to this state. Invariants play the leading part in optimization and it may be explained by the following simple examples. Let v be the invariant of the state a, $(v \Rightarrow u) = 1$ and schema S is deterministic (for any two transitions $a \xrightarrow{u/y} b$ and $a \xrightarrow{u'/y'} b'$ that flow from the same state a it is necessary that $u \& u' = 0$). Then transition $a \xrightarrow{u/y} b$ may be replaced by $a \xrightarrow{y} b$ and all other transitions from a may be deleted. Another example: if $\texttt{x} = \texttt{y} + \texttt{z}$ is invariant of the state a then the statement $P = (\texttt{p:=y+z-q})$ in the transition $a \xrightarrow{u/P} b$ may be replaced by p:=x-q. The methods of finding invariants of programs were studied in [3, 4].

The transformation which is called generalized loop unfolding consists of two stages — copying of states and changing transitions. Let $A_0 \subset A$ be some set of states of the U-Y-schema $S = (A, T)$. Add to A the set $B^{(n)} = A_0^{(n)} \cup \ldots \cup A_{n-1}^{(n)}$, supposing $A_i^{(n)} = \{(a, i) \mid a \in A_0\}$ and identifying the elements of $A_0^{(n)}$ with correspondent elements of the set A_0 $((a, 0) = a)$. For each of the transitions $a \xrightarrow{u/y} a'$, which flow from the states $a \in A_0$ and for any pair (a, i), $i = 1, \ldots, n-1$ add the transition $(a, i) \xrightarrow{u/y} (a', i)$, if $a' \in A_0$, or $(a, i) \xrightarrow{u/y} a'$, if $a' \notin A_0$. The stage of copying is finished.

On the stage of changing transitions any transition like $(a, i) \xrightarrow{u/y} (a', i)$ may be changed to $(a, i) \xrightarrow{u/y} (a', j)$. It is obvious that the program obtained as a result of generalized loops unfolding is equivalent to the source program. Especially if A_0 is a loop and for any i index j is chosen in accordance to the formula $j = (\texttt{if } i < n-1 \texttt{ then } i+1 \texttt{ else } n-1)$ or to the formula $j = i + 1 (\texttt{mod } n)$, then the analogs of well known transformations of while-loops will be obtained:

$$\texttt{while } \alpha \texttt{ do } P = (\texttt{if } \alpha \texttt{ then } P)^n \\ (\texttt{while } \alpha \texttt{ do } P),$$

$$\texttt{while } \alpha \texttt{ do } P = \texttt{while } \alpha \texttt{ do} \\ (\texttt{if } \alpha \texttt{ then } P)^n.$$

Loops unfolding makes it possible to do additional op-

timization after simplification of source program. The fact is that after unfolding the loops there appear new invariants which help to make additional optimization. Roughly speaking the more invariants the more possibilities for optimization. Therefore changing transitions in the loops unfolding must be done so as to obtain as more invariants as possible at the inputs of the loop body copies. This is the main criterion for optimal unfolding the loops. The considerations given here are of course too general and have mainly methodological character. But in the case of rewriting strategies they can be specified and generates rather effective speedup method.

5 Loops unfolding in strategies

For simplicity let us consider the top-bottom one time strategy. At first let us eliminate the recursion by replacing it by the manipulation with current node pointer and stack to memorize returns via nodes of the graph representing the expression t. As a result the following procedure description for `apply_R` will be obtained:

```
apply_R:=proc(t)loc(u)(
  set_init(u,t);
  contn:=1;
  while(contn,
    appls(u,R);
    move(u)
  )
);
```

Procedures `set_init` and `move` provide going around the tree representing `t` one time top-bottom. Statement `move(u)` moves pointer `u` to the next node which has not been yet observed. The condition `contn` means that `u` did not return yet to the initial node and strategy must continue its work.

Let Q be the optimized sequence of statements corresponding to the residual program for `applr(u,`R`)`. The sequence Q contains `goto` statements to the labels `M` (successful application) and `N` (unsuccessful application). After substituting Q to `while`-loop of `apply_R` this loop may be transformed to:

```
M: Q;
N: move(u);
contn->goto M;
```

Now the loop unfolding may be applied to this sequence which will be denoted further as QR. Let $a_1 \longrightarrow$`M`, $a_2 \longrightarrow$`M`,..., $a_n \longrightarrow$`M` be all transitions to the state labeled by `M`. Find some sets of invariants $I(a_1)$, $I(a_2)$,...,$I(a_n)$ for the states $a_1, \ldots, a_n$. These sets include the following types of conditions:

`v(`y`)=`x`,`
x`=`z`,` $\neq z$`,`
`type(`x`)=`ω`,` $\neq \omega$`,`
x`=v(`y`),` `v(`y`)`
`u=`f`(v(`y_1`),v(`y_2`),...)`.

Here x has the form x`=arg(u,`$i_1, \ldots, i_k$`)`, `ART(`z`)` $= 0$, $y, y_1, y_2, \ldots$ are parameters of R. All such invariants can easily be found by analyzing of Q. Some invariants may be lost because of confluences in the point `N`. To avoid this loss one can duplicate statement `move(u)` before loops unfolding and different transitions $b_1 \longrightarrow$`N`, $\ldots, b_m \longrightarrow$`N` change to $b_1 \longrightarrow$`N`$_1, \ldots, b_m \longrightarrow$`N`$_m$. Sequence QR must be transformed correspondingly to:

```
M: Q;
N1:  move(u);
contn->goto M else return;
.............
Nm:  move(u);
contn->goto M else return;
```

To unfold QR let us consider the set $\{E_0, I(a_1), \ldots, I(a_n)\} = \{E_0, \ldots, E_{p-1}\}$, E_0 being the set of invariants of the state `M` (generally speaking $p \leq n$, because some of $I(a_1), \ldots, I(a_n)$ may be equal). To compute invariants for transitions after the statement `move(u)` note that if `u` points to s and `ART(`s`)` > 0 then after `move(u)` `u` will point to `arg(`s`,1)`. Therefore for instance the invariant of the type `arg(u,1,`i`) = arg(u,1,`j`)` of the point `N`$_k$ will be transformed to `arg(u,`i`) = arg(u,`j`)`.

Introduce now p copies $QR_0, \ldots, QR_{p-1}$ of QR ($QR_0 = QR$), and change transitions $(a_i, j) \longrightarrow$(`M`, j) to $(a_i, j) \longrightarrow$(`M`, k) if $I(a_i) = E_k$. Now initial states of p copies of QR (`M`, 0), ..., (`M`, $p-1$) shall have invariants from $E_0, \ldots, E_{p-1}$ correspondingly and these copies may be simplified. After loops unfolding, computing invariants and simplifications it is necessary to identify the equivalent states to decrease the complexity of program.

New program may be considered as new loop and may be repeatedly unfolded. Note that this process can not continue infinitely because for given relation system R the set of all possible invariants of the types mentioned above is finite.

Let us return to the example considered in previous paragraph. After substituting optimized version of Q to the strategy `apply_R` and obvious changings the following program will be obtained:

```
M:(type (u)!=(()=>())) -> goto N1;
  x1--> arg(u,1);
  type(x1)!=(()&()) -> goto N2;
  x11--> arg(x1,1);
  type(x11)!=(()||()) -> (
    v(a) --> x11;
    v(b) --> arg (x1,2);
    v(c) --> arg (t,2);
```

```
    t:= ((v(a),v(b)) => v(c) );
    {E3} goto M;
  );
  v(a) --> arg(x11,1);
  v(b) --> arg(x11,2);
  v(c) --> arg(x1,2);
  v(d) --> arg(u,2);
  u:=(v(a)&v(c)=>v(d))&(v(b)&v(c)=>v(d));
  {E4} goto M;
N1:move(u);{E1}
  contn->goto M else return;
N2:move(u);{E2}
  contn->goto M else return;
```

Sets of invariants corresponding to four transitions to `M` are put into the figure brackets. The main invariants that may be used for simplification are:

```
u=((v(a),v(b))=>v(c))
```

from the set E_3 and

```
u=(v(a)&v(c)=>v(d))&(v(b)&v(c)=>v(d))
```

from the set E_4. Two steps of loops unfolding may be done productively. The final result is the following:

```
M:(type(u) != (()=>())) -> goto N1;
  x1--> arg(u,1);
  type(x1) != (()&()) -> goto N2;
  x11--> arg(x1,1);
  type(x11) != (()||()) -> (
    v(a) --> x11;
    v(b) --> arg (x1,2);
    v(c) --> arg (u,2);
    u:= ((v(a),v(b)) => v(c) ); goto M1;
  );
  v(a) --> arg(x11,1);
  v(b) --> arg(x11,2);
  v(c) --> arg(x1,2);
  v(d) --> arg(u,2);
L:u:=(v(a)&v(c)=>v(d))&(v(b)&v(c)=>v(d));
  goto M2;
M1:move(u);move(u); goto M;
M2:move(u);
M3:x11-->v(a);
  type(x11)≠(()||())->(
    u:=((v(a),v(c)) => v(d)); goto M1;
  );
  v(a)-->arg(x11,1);
  v(b)-->arg(x11,2); goto L;
N1:move(u);
  contn->goto M else return;
N2:move(u);
  contn->goto M else return;
```

System `R` is right linear, therefore the statements `set_init(v)` are redundant and may be eliminated from the program.

6 Implementation

The methods described in the paper is now being implemented in "Strategy design" subject domain in APS-1 system. The previous experiments shows that 2–10 time speedup may be achieved by means of loops unfolding. The choice of strategies is supported by programs that analyze relation systems. Formal transformations of the programs are supposed to be performed on APLAN level, the decisions being controlled by the user. The optimized strategy will be translated to C and than called as internal procedures.

References

[1] D. Bjorner, A. P. Ershov, and Jons, editors. *Partial Evaluations and Mixed Computations*, Proc. of the IFIP TC-2WS, North-Holland, 1986.

[2] P. Lescanne. Current trends in rewriting techniques and related problems. In *Trends in Computer Algebra*, number 296 in LNCS, pages 38–51. 1988.

[3] A. A. Letichevsky. On finding invariant relations of programs. In *Algorithms in Modern Mathematics and Computer Science (Urgench, 1979)*, number 122 in LNCS, pages 304–314, 1981.

[4] A. A. Letichevsky. On the possibility of completing a search for invariant equalities in programs. *Soviet Math. Dokl.*, 37(2):559–561, 1988.

[5] A. A. Letichevsky. Mixed computations and optimization of programs. *Programming*, (1):69–76, 1990. (In Russian).

[6] 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.

[7] A. A. Letichevsky, J. V. Kapitonova, and S. V. Konozenko. Algebraic programming system APS-1. To be published in TSI special issue.

[8] A. A. Letichevsky, J. V. Kapitonova, and S. V. Konozenko. Algebraic programming system APS-1. In O. M. Tammepuu, editor, *INFORMATICS '89, Proc. of the Soviet-French Symp.*, pages 46–52, Tallinn, May 1989. Institute of Cybernetics, Estonian Acad. of Sciences.

[9] M. J. O'Donnell. Term rewriting implementation of equational logic programming. In *Rewriting Techniques and Applications*, number 256 in LNCS, pages 1–12. 1987.

[10] P. W. Purdom and C. A. Brown. Fast many-to-one matching algorithm. In *Proc. 1st Conf. Rewriting Techniques and Applications*, LNCS, pages 407–416. Springer, May 1985.

[11] 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.