Tools for solving problems in the scope of algebraic programming

Y.V. Kapitonova, A.A.Letichevsky

November 29, 2001

Abstract

Algebraic programming system APS is considered as a tool for integrating computer algebra with artificial intelligence. The system is based on rewriting rule programming and algebraic program in APLAN, the source language of the system, in many cases may be considered as an executable specification of a problem. Two different kinds of solvers are specified in terms of rewriting rules. The first one is a universal solver that extends a pure PROLOG-like solver in different directions. One of the important property of this solver is the possibility for inclusion of special algorithms for solving equations in different algebras. Another solver is directed to solving problems on computational models (some kind of constraint networks). It searches for the solution of a problem in two stages - constructing the plan and solving equations. On the second stage the solver calls the universal one to get the solution of equations. The application of APS and its solvers to the development of system for mathematical education in secondary school is briefly described in the last section of the paper.

1 Introduction

Rewriting technique [1] is nowadays of a great interest as a reach field of investigations for theoretical computer science as well as a new information technology for practical use in many applications. Well known conferences such as [2], [3] consider different aspects of this field. There are a lot of systems which use rewriting as a basis for symbolic computation. The languages of OBJ family [4] and O'Donnell's languages [5] use rewriting as a basis for equational programming. ASF [6] and ASSPEGIQE [7] use rewriting as a basis for algebraic specifications. Rewriting is also used to extend the possibilities of systems based on other programming paradigms. Computer algebra systems such as Reduce [8] or MATEMATICA [9] are the examples.

Usually rewriting systems are considered as an extension of functional definitions and different restrictions such as regularity or confluence are supposed. For canonical term rewriting systems an arbitrary strategy of rewriting terminates and defines the unique result. The algebraic programming system APS [10], developed

in Glushkov Institute of Cybernetics, is based on rewriting but, in distinction from traditional approach, it is possible in APS to combine arbitrary systems of rewriting rules with different strategies of rewriting which may be specified explicitly by means of procedural definitions. The separation of equational definitions in the form of rewriting rules from the strategy of rewriting essentially extends the possibilities of rewriting technique enlarging the flexibility and expressibility of it. The APS integrates four main programming paradigms (procedural, functional, algebraic and logic) by adjusted use of corresponding computational mechanisms as it was described in [10]. Procedures are used in the system to define rewriting strategies, functional definitions are special case of algebraic ones and logic programming is realized by defining strategies of solving problems on axiomatically defined subject domains in terms of rewriting.

The mentioned features of APS make it to be a good base for integrating computer algebra and artificial intelligence. There are many successful examples of such an integration nowadays. `Scratchpad/AXIOM` ([20], [17], [18]) based on the notions of domains and categories has been developed for description of algebraic structures. `AXIOM` includes reach hierarchy of classical mathematical domains and efficient algorithms for solving problems for them. Another example of integration of database and deductive facilities with computer algebra algorithms is system called `Cayley/MAGMA` [23] for computations in the group theory.

The hybrid knowledge representation system `MANTRA` [19] is a next step of such integration. It combines different formalisms for specification of mathematical domains and development the computational environment for solving problems combining the strong mathematical algorithms with heuristic search for solutions.

On the other hand, many famous large computer algebra system are developing for usage in the intelligent framework. The `Praxis` [21] system is implemented as intelligent user interface for the symbolic algebra system `Macsyma`, using a rule-based expert system. `Reduce` theory of computational domains [24] is attempt to design the formal means for description of interaction among `Reduce` algorithms.

Very important is the problem of programing language for efficient development of computation algorithms as deductive tools. This language must provide a semantics which is able to combine different programing paradigms. For example, the language `LIFE` [22] integrates the logic and functional programing, equations and inheritance. It allows combine database, reasoning, simplification and function evaluation. Many efforts was made also in the scope of constraint logic programming paradigm [27].

In spite of these examples the problem of efficient combination of computer algebra algorithms and searching methods from artificial intelligence has not yet satisfactory solution. The investigations continue and the development of programming tools for their support is actual.

In this paper two different kinds of solvers are specified in terms of rewriting rules. The first one is a universal solver that extends a pure PROLOG-like solver so, that arbitrary predicate formulas may be used in the right hand sides of clauses and left hand side may be the negation of an atomary formula. The queries are arbitrary predicate formulas (without quantifies). The semantics of the solver may

be defined in terms of three-valued logic as it has been done in [13]. One of the important property of this solver is the possibility for inclusion of special algorithms for solving equations in different algebras. This possibility is illustrated by including linear equations over fields.

Another solver is directed for solving problems on computational Tyugu models. This concept close to constraint networks of today [16] has been developed in 70-th for specifications of engineering problems and used in a problem solver Utopist [15]. The computational model solver searches for the solution of a problem in two stages - constructing the plan and solving equations. On the second stage the solver calls the universal one to get the solution of equations.

The application of APS and its solvers to the development of system for mathematical education in secondary school [25] is briefly described in the last section of the paper [25]. This system is intended for supporting mathematical education process. It is able to solve algebraic and trigonometric problems as simplifications, proofs of identities and solutions of equations. APS was used for the development of the mathematical kernel of the system.

2 Introduction to APS

In this section some information about APS and its source language APLAN is presented. This information is necessary to understand next sections and some important features of algebraic programming technology used in APS. More complete description of APS may be found in [10].

Algebraic programs written in APLAN are located in algebraic modules (ap-modules). Each module contains description of names, description of operation symbols (marks) with corresponding syntactical information and initial assignments for names. Data structures are presented by terms, generated by operations starting with numbers, names and atoms (symbols which cannot have values). Special types of terms are equalities, which may represent rewriting rules, rewriting systems, procedures and statements which are parts of procedures. Any procedure in an ap-module may be called and performed by one of the system interpreters.

The following example of ap-module named `pl_ac.ap` will be used to explain these notions more precisely.

```
INCLUDE <rat.ap>
/*
      Expanding polynomials represented in natural form
*/
NAMES rdn, canpl;
NAMES pw,pow,bn;
/*
                    Specification of canpl
*/
```

```
canpl:=proc(t)(can_ord(t,rdn,rdn));

rdn:=rs(q,x,y,z,u,k,n,a)(

     isnum(x) -> (x*y = y$x),
     isnum(y) -> (x*y = x$y),
     isnum(x) -> (x$y = y*x),

                 x - y = x+(-1)*y,

                 x $ 0 = 0,
                 x $ 1 = x,
           (x + y) $ z = x $ z + y $ z,
           (x $ y) $ z = x $ (y * z),

               (x+y)*z = x*z+y*z,
               x*(y+z) = x*y+x*z,

               (x$y)*(z$u) = (x*z)$(y*u),
               (x$y)* z    = (x*z)$y,
                x   *(y$z) = (x*y)$z,

               (x$y)^n = x^n$y^n,
               (x*y)^n = x^n*y^n,
               (x^y)^n = x^(y*n),
               (x+y)^n = pw((x+y)^n)

);


pw:=proc(t)(
    can_ord(t,pow,rdn);
    return(t)
);

pow:=rs(x,y,z,n,k,q,a)(
    n>1 ->   ((x+y)^n = bn(1,x,y,1,n,n) +   y^n),
    n>1 -> (q*(x+y)^n = bn(q,x,y,1,n,n) + q*y^n)
);

bn:=rs(q,x,y,k,n,a)(
    (q,x,y,n,n,a) = q*x^n,
    (q,x,y,k,n,a) = (x^k*q$a)*y^(n-k)+bn(q,x,y,k+1,n,(a*(n-k))/(k+1))
);
```

```
NAME T;

T:=((a+b)*(b+1/2)+(a-5*b)*(b+c))*(a+b+c)*(b+c-d);

task:=(
    canpl(T);
    prnpl(T)
);
```

The first sentence of the module includes to it previously designed module `rat.ap` which especially defines all operations used in `pl_ac.ap`, including arithmetical operations, separators (”,” and ”;”), and so on. It also defines operations over rational numbers and indicates that ”+” and ”*” are associative-commutative operations, inserting them to the data structure called `ac_list`. This information is used by some rewriting strategies.

The module specifies an algorithm which expands a polynomial with numerical coefficients reducing it to the natural canonical form - the sum of monomials. The operation $ is used to denote the multiplication of polynomial by number to distinguish it from the multiplication ”*” of polynomials. Any symbol (atom or name) is considered as a variables of the polynomial.

Initial values of the names `rdn, pow` and `bn` are rewriting systems. Rewriting rules are represented by equalities and conditional equalities. The application of rewriting system to a term is made in the following way. Rules are matched with the top operation in the order they are written in the system. The first matched rule is applied to the top operation of a term. The matching is made in free algebra, but after substituting the right hand side of a rule and the values of variables (listed in the head of a system) which they got when matching, a new part of a term is reduced to the *basic canonical form*. This reduction is top-bottom application of interpreters for interpreted operations if there are any. The interpreters are built-in for some operations, or may be defined by user. The arithmetical operations and relations are interpreted on numbers in a usual way with some additional simplifications, such as $x + 0 = x$ or $x * 1 = 1$. Conditional rules are applied only if after matching the basic canonical form of a condition is 1.

To apply rewriting system to the term different strategies may be used. Each strategy defines some movement along the nodes of a tree in searching for the subterms to which the system is applicable, and some additional transformations defining implicitly applied rewriting rules or simplifications. Simple example of strategy is strategy `ntb(t,R)`. It applies the system `R` to the term `t` searching for mentioned subterms (redexes) top-bottom and left right. This is a lazy strategy. Dual to it is the bottom-up strategy `nbt(t,R)` which corresponds to call-by-value. In the considered case more complicated strategy can_ord(t,R1,R2) is used. This strategy works with two systems of rewriting rules. First system is applied top-bottom, second - bottom-up. At that when the strategy moves over the nodes bottom-up the subterms are ordered w.r.t. ac-operations by means of merging already ordered arguments of such operations.

Each strategy may be specified as a simple recursive procedure in APLAN in terms of two basic strategies `applr(t,R)` and `appls(t,R)` realized on the low level of APS (C language). The first one applies the system `R` to the top operation of `t` one time, the second repeats the application of `R` while possible. Here the special role of application operation must be mentioned. This operation is interpreted. If `x` is a function or the name of a function than this function is applied to its argument `y` when the expression `x(y)` is reduced to the basic canonical form. There are two types of functions in APLAN - rewriting systems and procedures. If the function is a procedure, the strategy `applr` is used to compute the value of a function. The name of a function may occur in the right hand side (as in the definition of system `bn` in the example above). In this case the function is computed recursively. Therefore `applr` is called recursive strategy and `appls` iterative or recursive-iterative one.

Now let us consider specification of the strategy `can_ord`:

```
NAME can_ord,can_up;

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)
);
```

Function `can` calls the reduction to basic canonical form, `arg(t,i)` is `i`-th argument of `t`, in the loop `forall(s=arg(t,i),...)` `i` varies from 1 to the arity of `t`. Procedure `merge(t)` merges two arguments of `t` if the top operation of `t` is associative-commutative operation.

Data structures in APS really are graph terms although the initial representation of algebraic expressions, occurring in APLAN program are trees. Therefore APLAN has two different kind of assignments `x-->y` and x:=y. The first one sets the name

x, considered as a pointer to the top node of the data structure, which is the value of y, the second one curries the copy of the top node of y with all its references to the place where the current value of the name x is placed. The exact definition of the loop `forall` may be expressed by means of simple for statement:

```
for(i:=1,i<=ART(t),i:=i+1,
    s-->arg(t,i);
    <body of the loop forall>
)
```

where `ART(t)` is the arity of `t` and the function `arg` returns not copy but the original top node of `i`-th argument of `t`.

The simplicity of `can_ord` makes it easy to prove the correctness of the procedure `canpl(t)` using induction on the size of the term `t` after exact definition of what is canonical form of a polynomial.

3 Universal solver

The solver described below searches for the solution of a problem on subject domain defined by the set of axioms, which are quantifierless formulas with variables assumed to be tied by universal quantifier. Each formula is supposed to be elementary (that is atomary formula or the negation of atomary formula) or represented as implication $P \Rightarrow Q$ where P is arbitrary formula, Q is elementary one. The solver will use PROLOG-like strategy for solving problems and each axiom in the form of implication will be used as PROLOG clause or inference rule that reduces problem to subproblems.

The signature of predicates may contains equality. The axioms define some equational theory for it. It consists with all equalities which are the sequences from the set of axioms. The solver may use some special algorithms for solving equations in this theory without referring to axioms. The same may be said about some other predicates.

An absolutely free algebra of terms of a given signature, generated by the set of constants A and the set of variables Z, will be denoted as $T(A, Z)$ (initial algebra with the set $A \cup Z$ of operations of arity 0). The variables of axioms are assumed to belong to the set $W = \{W(1), W(2), \ldots\}$.

An *elementary problem* is a pair (P, X), where P is an arbitrary predicate formula without quantifiers and with free variables from the set $V = \{V(1), V(2), \ldots\}$ and X is a V-*context* with values in $T(A, V)$ that is a substitution of a type $\{V(1) \leftarrow t_1, \ldots, V(n) \leftarrow t_n\}, t_1, \ldots, t_n \in T(A, V)$. A *solution* of an elementary problem (P, X) is a new V-context $Y = \{V(1) \leftarrow s_1, \ldots, V(n) \leftarrow s_m\}$ such, that $m \geq n, s_i$ is an instance of t_i for $i = 1, \ldots, n$ and PY is a consequence of axioms. What means a consequence exactly is defined by operational (calculus) or denotational (set - theoretical) semantics of the language.

When the problem is being solved, the set of subproblems appears, and we may speak about the *complex problems*, that correspond to the sets of elementary

problems and their solutions. Syntactically, the general notion of a problem is defined as follows.

1. Elementary problem is a problem;
2. Context is (a solved) problem or a solution;
3. `fail` is (unsolvable) problem;
4. If P and Q are problems, than $P \mid Q$ is a problem;
5. If P is a formula, Q is a problem, then (P, Q) is a problem.

Intuitively, solution of a problem $P \mid Q$ is a solution of P or a solution of Q, solution of (P, Q) is a solution of (P, X) where X is one of the solutions of Q. Therefore a notion of (undecidable) problem corresponds to a search AND-OR-tree. If the problem P is to be solved using special algorithms for solving equations or predicates, the name z of corresponding algebra must be joined to the problem. Problem $z(P)$ is called *specialized*.

Operational semantics of the language is defined by means of the partial function `solve` which maps problems to problems. The main property of this function is that `solve`$(P) = X$, where X is one of the solutions of the problem P (if there are any) or `solve`$(P) = X|Q$, where X is one of the solutions and one may obtain another solutions applying `solve` to Q. The best situation is when all solutions may be covered by this process. Function `solve` applies the system `solve_rs` to a problem with iterative strategy `appls`. Therefore the rewriting rules of this system may be considered as inference rules of corresponding calculus. The problem to be solved must be specialized. If there are no special algorithms for the problem it must be specialized by the name `fr` of free algebra. Following are the definitions in APLAN.

```
solve:=proc(p)(
    appls(p,solve_rs);
    return(p)
);
solve_rs:=rs(P,Q,R,X,Y,a,b,x,y,z)(

                fail|Q =  Q,                          /*  1 */
               (P|Q)|R = P|Q|R,                       /*  2 */
    is_not_sol(x)->(x|y = solve(x)|y),                /*  3 */

      z(    P,    fail) = fail,                       /*  4 */
      z(~(~(P))     ,X) = z(P,X),                     /*  5 */
      z(~(  P &  Q ),X) = z(~(P) || ~(Q),X) ,         /*  6 */
      z(~(  P || Q ),X) = z(~(P) &  ~(Q),X),          /*  7 */

      z(    P &  Q  ,X) = z(Q,solve(z(P,X))) ,        /*  8 */
```

```
      z(    P || Q  ,X) = z(P,X) | z(Q,X) ,              /*  9 */

            z(P,  Q|R ) = z(P,Q)| z(P,R),                /* 10 */
            z(P,y(Q,X)) = z(P,solve y(Q,X)),             /* 11 */

      z(P,(a,Y)|-(Q,X)) = z(P,solve((a,Y)|-(Q,X)) ),     /* 12 */

      z ( (P = Q),   X) = (vl(z).solve_eq)((P = Q),X),   /* 13 */
      z (  P,        X) = (vl(z).solve_pr)(P,X),         /* 14 */

      ((a, b),Y)|-(P,X) = (a,Y)|-(P,X) | (b,Y)|-(P,X),   /* 15 */
      ( R=>Q, Y)|-(P,X) = try (R,ART(X),unf(P=Q,X,Y)),   /* 16 */
      ( a,    Y)|-(P,X) = unf(P=a,X,Y)                   /* 17 */


);

is_not_sol:=proc(x)(return(~(mark(x)==mark_ar)));


try:=rs(R,n,X)(
    (R,n,fail) = fail,
    (R,n,X  )  = fr(rename(R,n,X),X)
);
```

First 11 rules deals with general problem and reduce it to elementary one. Sign `||` is disjunction. Rules 13 and 14 solve elementary problems referring to the algebra that specify it. The name of an algebra in specialized problem is the name of data structure called `valuation`. Components of a valuation are named by atoms and referred to by means of operation `z.x` which denotes the body of a component with the name `x` of a valuation `z`. Valuation for an algebra must contain at least two names: `solve_eq` and tt solve_pr for algorithms to solve equations and predicates, correspondingly. The description of free algebra is the following:

```
fr:=(
    solve_pr: rs(P,z,x,X)(
        (~(P(x)),X)  = (vl(P).neg,make_nil(vl(P).head))|-(~(P(x)),X),
        (  P(x), X)  = (vl(P).pos,make_nil(vl(P).head))|-(  P(x), X)
    );
    solve_eq: rs(P,Q,X)(
        ((P = Q),X) = unify((P = Q),X)
    )
);
```

Function `solve_eq` for this algebra call unification procedure, realized on low level. It returns new context which represent the most general unifier of `P` and `Q` or symbol `fail` if unification of the terms is impossible. For solving predicates (elementary formulas) function `solve_pr` refers to definition of a predicate. Noninterpreted predicate symbol is the name of a valuation where all axioms related to this predicate are differed on two groups: positive and negative. Following is the example of predicate definition:

```
P1:=ax(x,y,z)(
    pos: (
        P1(A,B),
        P1(x,A+x),
        P1(x,y)|| ~(P2(y,z) => P1(x+z,y)
    ),
    neg:(
        ~(P1(C,D)),
          P3(y,y)=> ~(P1(x,y,z))
    )
);
```

This is the external representation. The head of this definition contains the list of variables which must be translated to `W(1),W(2),...`. The head is also used for creating the empty context (function `make_nil`). Function `solve_pr` returns new type of a problem: *inference problem*. It has a pattern $A \| B$ where A is a list of axioms and B is an elementary problem. This kind of a problem is considered by rules 15-17. Function `un` is the modification of unification for two contexts, the solution (new context) may contain two kind of variables - axiom variables and problem ones. The function `rename` renames axiom variables if any to new problem ones.

The contexts are represented as one-dimensional arrays and the condition `is_not_sol(x)` (is not a solution) in the rule 3 checks if the type of `x` is array. This rule shows that the search for the solution is depth-first-search. To get breadth-first- search or parallel strategy, this rule must be modified to `is_not_sol(x)->(x|y = solve(y)|x)` and a function that make only some steps of solving problem must be called from solve_rs instead of `solve`.

Another example of an algebra is the algebra `lin_alg` that contains an algorithm for solving linear equations over field:

```
lin_alg:=(
    solve_pr:rs(P,X)((P,X) = (vl(fr).solve_pr)(P,X));
    solve_eq: proc(p)(
        canpl(p);
        yes:=1;
        appls(p,solve_lin_rs);
```

```
        ntb(p,del_mlt);
        return(p)
    )
);
```

To solve predicate the algebra refers to free case. The procedure `canpl` is modified so that symbols different from unknowns `V(i)` would be considered as constasnts and be included to coefficients. Rewriting systems used in `solve_eq` are the following.

```
solve_lin_rs:=rs(A,B,E,X,i)(

    (V(i)$A+B = 0,V(i)=nil,X) = starg(X,i,canplf((-1)*(1/A)*B)),
    (V(i)  +B = 0,V(i)=nil,X) = starg(X,i,canplf((-1)*      B)),
    (V(i)$A   = 0,V(i)=nil,X) = starg(X,i,0),
    (V(i)     = 0,V(i)=nil,X) = starg(X,i,0),

    (E,           V(i)=A,  X) = (canplf(sub(V(i)=A,E)),X),

    (0=0,X) = X,
    (A=0,X) = make_sys(A=0,unknown(A),X),
    (A=B,X) = (canplf(A+(-1)*B=0),X)
);

make_sys:=rs(E,i,X)(
    (E,nil,X) = fail,
    (E,  i,X) = (E,V(i)=arg(X,i),X)
);

unknown:=rs(A,B,i)(
    A+B = unknown(A),
    A$B = unknown(A),
    V(i)= i,
    A   = nil
);
```

Some technical explanations. Function `canplf` is functional modification of `canpl`. Function `starg(X,i,z)` updates the array `X` setting its `i`-th argument to `z`.

Semantics. Semantics of the solver, may be described in the terms of three-valued logic of Klinee on the base of the paper [12] as it was done in [13]. This logic is the logic of partially defined predicates, and its truth values $\perp, 0, 1$ correspond to undefined, false and true values. This set of values is assumed to be partially ordered so that $\perp \sqsubset 0, 1$ and $0, 1$ are not comparable. Let us fix some signatures of operations and predicates (it does not matter if these signatures are multisorted or one-sorted), and consider algebras with partially defined predicates of the given

signature. Gomomorphism is defined as a mapping which preserves the operations and is monotonous w.r.t. logical values of predicates. Equality is also considered as partial predicate which relates with real equality so that $(d = d\prime) = 1 \leftrightarrow d = d\prime$ (therefore if $d \neq d\prime$, then $(d = d\prime) = 0, \perp$).

Algebra D is called to be *labelled* by a set of constant symbols A, if the labelling mapping $v\colon A \to D$ is defined. If $d = v(a)$, the element d is called to be labelled by symbol a. Labelled algebra is supposed to be generated by all labelled elements.

Now let $\Phi(A, W)$ be a language of arbitrary predicate formulas without quantifiers with the set of constants A and variables from the set W, constructed by means of logical connectives $\vee, \wedge, \neg$. Formula $P(x_1, \ldots, x_n) \in \Phi(A, W)$ is called to be true on the algebra D, labelled by A, $(D \models P(x_1, \ldots, x_n))$ if it is true on D for all ground values of therms $x_1, \ldots, x_n$. Let F be a subset of $\Phi(A, W)$. An algebra D labelled by A is called to be a *model* of a set F if each formula from F is true on D. A set F of formulas is *consistent* if it has a model.

Let D and $D\prime$ be two algebras, labelled by the sets of constants A and $A\prime$, correspondingly. D is called to be an *approximation* of $D\prime$ if $A \subset A\prime$ and there exists a gomomorphism $\gamma\colon D \to D\prime$ such, that $\gamma(v(a)) = v\prime(a)$ for all $a \in A$, where v and $v\prime$ are the labelling functions for D and $D\prime$, respectively. Two algebras are called to be isomorphic if each approximates other. The approximation relation is a partial order on the class of all labelled algebras (of the same type) considered up to isomorphism. The minimal element in the class of all algebras, labelled by the same set A, is absolutely free algebra $T(A)$ of ground terms with nowhere defined predicates.

Algebra D labelled by a set A approximates a set F of formulas if it approximates every model of that set, labelled by a set $A\prime \subset A$.

The class of all approximations of a consistent set of formulas F, labelled by the same set A, has a maximal element, defined uniquely up to isomorphism.

The maximal approximation may be constructed as a factor-algebra $T(A)/F = T(A)/\rho(F)$, where $\rho(F)$ is the congruence, defined by relation: $t = t\prime(rho(F)) \Leftrightarrow F \models t = t'$. For predicates define $p(t_1, \ldots, t_n) = 1 \iff F \models p(t_1, \ldots, t_n)$ and $p(t_1, \ldots, t_n) = 0 \iff F \models \neg p(t_1, \ldots, t_n)$. Therefore, if no one of two alternative is true, $p(t_1, \ldots, t_n) = \perp$. For detailed proof of this theorem and theorems mentioned below see [13]. Maximal approximation is not generally a model of F, but it contains a complete information about all models of F: formula $P \in \Phi(A, X)$ is true on all models of F if it is true on $T(A)/F$. The consistency of F for three-valued logic is the same as the classical consistency. Really, let us denote $(F)^=$ the extension of F by the axioms of equality $(x = x, x = y \Rightarrow y = x, x = y \wedge P(x) \Rightarrow)$. Than F is consistent if $(F)^=$ is classically consistent (that is has two-valued model). This statement is the consequence from the theorem below. The model of F is called to be complete if every ground atomary formula has the value different from $\perp$.

The set F is consistent if it has a complete model.

The theorem and its corollary provides the possibility to use the ordinary resolution calculus with paramodulation as a complete deductive system for inference of all semantical consequences from the given set F of formulas. Indeed, every logical formula in Kleene logic may be reduced to conjunctive normal form and F may be as-

sumed as a set of disjuncts. Then for any atomary formula P, $F \models P \Leftrightarrow (F)^{=} \cup \{\neg P\}$ is classically inconsistent, but it means that P is inferred from F in the calculus of resolution with paramodulation. Now to set the connection of solver, described above, with this deductive system is the matter of well known technique.

The solver is obviously consistent, that is every solution provided by it is the solution on a maximal approximation. But it is not complete (even if there are no specialized predicates and equations) for several reasons. The first is that solver uses SLD-resolution inference which is known to be incomplete in the case of non-Horn clauses. So the full resolution calculus must be modeled by solver. The simplest (but not the best) way is to extend the set of statements of evaluated program by intermediate results at each step of inference. Another reason is that the solver realizes the depth-first-search. It may be transformed to the breadth-first search device (less efficient) and than it will be complete if there are no axioms, containing equality (the maximal approximation in this case is absolutely free algebra). To make it complete in any case, the rules for paramodulation must be added.

4 Problem solving on computational models

Computational model is a set of variables (elements of a model) and the set of relations or constraints, binding possible values of these variables. Special type of constraints is algebraic equations. A problem on computational model is a question of a type: "find the values of $y_1, \ldots, y_m$ assuming the values of $x_1, \ldots, x_n$ to be known". Known values may be given as a constants of subject domain, defined by a model, or symbolically. The last case may be recognized as a special type of a problem of program synthesis. Example of computational model (in terms of APLAN data structures):

```
triangle:=comp_model(

    elements:(a,b,c,alpha,beta,gamma);

    equations:(

       alpha+beta+gamma=pi,

       b^2=a^2+c^2+2*a*c*cos(beta),
       c^2=a^2+b^2+2*a*b*cos(gamma),

       a/sin(alpha)=b/sin(beta),
       b/sin(beta) =c/sin(gamma),
       c/sin(gamma)=a/sin(alpha)
    );

    solutions:........;
```

```
    solve_eq:.........;
    solve_pr:.........;
    ...................
}
```

Each relation may be considered as a sours for finding solution of *elementary problem*. For instance, the equation `b^2=a^2+c^2+2*a*c*cos(beta)` may be used for finding `a` if `b,c,beta` are known, or for finding beta if `a,b,c` are known. Let us suppose that for each element of a model the set of all possible elementary problems are given, say, in the form (for example above):

```
    solutions:(

       a:(
         b c alpha     compute(a: a^2=b^2+c^2+2*b*c*cos(alpha))||
         b c beta      compute(a: b^2=a^2+b^2+2*a*b*cos(beta))||
         b c gamma     compute(a: c^2=a^2+b^2+2*a*b*cos(gamma))||
         b alpha beta  compute(a: a/sin(alpha)=b/sin(beta))||
         c alpha gamma compute(a: c/sin(gamma)=a/sin(alpha))
       );

       b:(
         a c alpha     compute(b: a^2=b^2+c^2+2*b*c*cos(alpha))||
         a c beta      compute(b: b^2=a^2+c^2+2*a*c*cos(beta))||
         a c gamma     compute(b: c^2=a^2+b^2+2*a*b*cos(gamma))||
         a alpha beta  compute(b: a/sin(alpha)=b/sin(beta))||
         c beta gamma  compute(b: b/sin(beta) =c/sin(gamma))
       );
       ........................
    )
```

These solutions may appear as a result of preliminary analysis of equations of a model, or a procedure which generates possible solutions for each element of a model may be put as a value of the name `solutions` to be used in run time. Following is the rewriting system for making the plan for solving a problem on computational model. It is used with the iterative strategy and transforms a problem to the plan of its solution. A problem is given in the form `problem x => y` where `x` is the list of known elements `y` the list of unknowns. Elements in a list are separated by applications (blanks).

```
make_plan:=rs(P,Q,R,Q1,Q2,F,x,y,z,u)(

                problem(x => y) = (e,x => y e, e) || no_solutions,
```

```
              (Q1 || Q2) || Q  =  Q1 || Q2 || Q,
            no_solutions || R  = R || no_solutions,
     (P,x => e,         e) || R  =  simp(P),
     (P,x => (Q1  Q2)Q, z) || R  =  (P,x => Q1 Q2 Q, z) || R,
     (P,x => (Q1||Q2)Q, z) || R  =  (P,x => Q1 Q, z)||(P,x => Q2 Q, z) || R,

     (P,x => compute F Q, y z) || R = (P F, y x => Q, z) || R,

 is_in(y,x) ->( (P,x => y Q, z)  || R = (P y,   x => Q, z) || R ),
 is_in(y,z) ->( (P,x => y Q, z)  || R =  R ),

        (P,x => y Q, z) || R  =  R || (P,   x =>  (get_sol y) Q, y z)
);

simp:=rs(x,y,z)(
       e x    =   simp(x),
      (x y) z =   simp(x y z),
       x y    = x simp(y)
);
```

Function `get_sol(y)` calls the solutions for the element `y`. It may return, for instance, `vl(model_name).solutions).y` where `model_name` is a global name which contains the name of a model. Intermediate data structure `(P,x => Q,z)` which appears in the left hand sides of rewriting rules, has the following meaning. `P` is already made part of a plan, `x` - list of known elements, list of unknowns with first element possibly changed to the disjunction of possible solutions, `z` is the list of elements for which it were calls to `get_sol` function in right-left order. If generalize the notion of problem to the form which appears in the rewriting rules, it may be proved that the rules preserve equivalence of problems, and each elementary problem may appear no more than once in the problem. Therefore the iterative strategy terminates and gives correct plan. It has linear time complexity w.r.t. to the number of elementary problems, and a plan cannot be simplified, there is no redundant computations.

In the example, considered above the problem `problem a b gamma => beta c alpha` will be rewritten to the plan

```
a b a b gamma (c : c ^ 2 = a ^ 2 + b ^ 2 + 2 * a * b * cos gamma)
(alpha : a ^ 2 = b ^ 2 + c ^ 2 + 2 * b * c * cos alpha)
gamma (beta : alpha + beta + gamma = pi) c alpha
```

To get the symbolic or numeric solution, the plan must be (automatically) converted to the problem for universal solver:

```
trn(
    V(2) ^ 2 = a ^ 2 + b ^ 2 + 2 * a * b * cos gamma &
    a ^ 2 = b ^ 2 + V(2) ^ 2 + 2 * b * c * cos V(3) &
    V(3) + V(1) + gamma = pi, array(nil,nil,nil)
)
```

and solved by special solver for computational model `trn`.

5 Application of APS to school mathematical education

A system for computer support of mathematical training in secondary school on the base of APS is now under development. The first version of this system, called AIST has been discussed in [25]. Second version of this system, called TerM (Terra Mathematica) has more powerful algorithms of solving equations and better theoretical background, based on well-defined algebraic hierrachy. The solvers, described in this paper, are included to the system and used for organization of computational processes on a base of different special algorithms.

The main task of the system `TerM` is support for algebraic and trigonometric problems on simplifications, proofs of identities and solutions of equations (further on the term "algebraic problem" is used namely for this type of problems).

The mathematical activity of a student consists of recognizing properties of mathematical objects and its transformations according to the rules, strictly defined in a corresponding mathematical theory. System either verifies transformation made by a student (`Short-Step` mode) or automatically executes transformation according to the students instruction (`Long-Step` mode).

Thus, the solution process of an algebraic problem is a sequence of steps. Every successive step is the result of some algebraic transformation of the previous one. The sequence steps from the setting up of the task and terminates with its solution.

Like other pedagogical computer systems, `TerM` provides the user by the mathematical `Reference Book.`

Windows shell `Copybook` is designed as a computer model of the student's copybook where student can solve a current problem, save and look through the solving of the previous problems. The natural notation of algebraic expressions in a schooltype syntax is provided with specialized formula editor.

`TerM` contains the subsystem `Solver` aimed at automatic solving of trigonometric problems. The algorithm is based on the equation-type classification (this approach is similar to the conception of the `PRESS` [26] system). The classification algorithms use canonical forms and hierarchy of algebras. This hierarchy especially the following algebras:

`Trig` - The field of rational trigonometric expressions whose arguments belong to `Arg` and coefficients are from `Coef` (sin, cos, tan, cot is a signature)

`Coef` - The Coefficients field

`Arg` - Vector's space of arguments

`Sol` - Algebra of Sets of arithmetic progression-solutions of a trigonometric equations

`BField` - The basic field of zero - characteristic. We use field of rational numbers

`BIntDom` - (Commutative) Integral Domain. Ring of integers

These algebras constitute the basis. Other algebras may be defined by recoursive typing with constructive definitions of gomomorphisms and inclusions.

6 Conclusion remarks

We have presented the basic ideas of problems solving in the algebraic programming environment and its application. This technique is effective for integrating computations and logic in the common framework. We are going to use these methods to develop automatic means for specializations of general algorithms on a base of partial evaluation with respect to rewriting rule systems. The general theory of computing invariants of programs will be used to develop the algorithms.

References

[1] N.Dershovitz, J.-P.Jouannaud, Rewrite systems. In*Jan van Leeuwen, ed. Handbook of theoretical computer science, v.B* Elsevier, 1990.

[2] C.Kirchner(Ed.), Rewriting techniques and Applications.Proceedings, LNCS vol. 690, 488, Springer,1993.

[3] M.Rusinovich,J.L.Remy(Eds.), Conditional term rewriting systems.Proceedings,1992, LNCS vol.656,Springer, 501, 1993.

[4] J. Gogen, C. Kirhner, H. Kirchner, A. Megrelis, and T. Wincler. An introduction to OBJ-3. In Jouannaud and Kaplan (Ed.). *Proc. 1st Intern. Workshop on Conditional Term Rewriting Systems*. Springer-Verlag, 1988.

[5] M. J. O'Donnell. Term rewriting implementation of equational logic programming. In P. Lescanne (Ed.) *Rewriting Techniques and Applications*, volume 256 of *LNCS*, Springer-Verlag, 1987.

, pages 1–12.

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

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

[8] Rayna G., REDUCE.Software for Algebraic Computation. N.-Y.,Springer,1989.

[9] S. Wolfram. *Mathematica*$^{\text{TM}}$. *A System for Doing Mathematics by Computer.* Addison-Wisley, 1988.

[10] A.A.Letichevsky,J.V.Kapitonova,S.V.Konozenko, Computations in APS, *Theoretical Computer Science* 119(1993),145-171, Elsevier,1993.

[11] Letichevsky A.A., Kapitonova J.V., Konozenko S.V Algebraic Programs Optimization. In*Proc. of the Int. Symp. on Symbolic and Algebraic Computation* (ISSAC'91), July 15-17 1991, Bonn, Germany, ACM Press, New York 1991

[12] M.Fitting, A Kripke-Kleene semantics for logic programs, *J.Logic Programming*, 4,1985,295-312.

[13] J.V.Kapitonova,A.A.Letichevsky, On constructive mathematical descriptions of subject domains, *Kibernetica*, 4,1988.

[14] J.M.Hullot, Canonical forms and Unification, In Proc.of 5-th Conference on Automated Deduction, LNCS v.87,318-334,Springer,1980.

[15] Enn Tyugu, Solving problems on computational models. J.Computational Mathematics and Math.Phys.,10:716-33, 1970.

[16] Ugo Montanary Networks of constraints: Fundamental properties and application to picture processing. *Information Sciences*, 7(2):95-132,1974.

[17] Davenport, J.H. and Trager, B.M. Scratchpad's View of Algebra I: Basic Commutative Algebra. *Lecture Notes in Computer Science.* (429),Springer-Verlag, 1990, pp. 40–55.

[18] Davenport, J.H., Gianni, P. and Trager, B.M. Scratchpad's View of Algebra II: A Categorial view of factorization. In*Proc. of the Int. Symp. on Symbolic and Algebraic Computation* (ISSAC'91), July 15-17 1991, Bonn, Germany, ACM , New York 1991.,pp.32–39

[19] Calmet J., and Tjandra, I.A. A Unified-Algebra-based Specification Language for symbolic Computing. In *Design and Implementation of Symbolic Computation System*, Springer-Verlag, 1993, pp. 14–27.

[20] Sutor, R.S. (Ed.) Axiom. *User's Guide.*, The Numerical Algorithm Group Limited, 1991.

[21] Clarkson, M. Praxis: rule-based expert system for Macsyma, *Lecture Notes in Computer Science.*,(429),Springer-Verlag, 1990, pp.264–265.

[22] Ait-Kaci, H. and Podelski, A. An overview of LIFE. *Lecture Notes in Computer Science.*,(504),Springer-Verlag, 1991, pp. 42–58.

[23] Butler, G. and Cannon, J.J., Cayley, version 4: the user language, *Lecture Notes in Computer Science.*,(358),Springer-Verlag, 1989, pp.456–466.

[24] Bradfort, B.J., Hearn, A.C., Padget, J.A. and Schrufer, E.. Enlarging the REDUCE domain of computation. In *SYMSAC 1986*, pages 100–106., ACM, New York, 1986.

[25] L'vov M.S., Kuprienko A.B. and Volkov V.A. Applied Computer Algebra System AIST: Computer Support of Mathematical Training, In *Proc. Int. Workshop on the Computer Algebra Application*,,July 9, 1993, Kiev, Ukraine.

[26] Silver, B. *Meta-level Inference.* Elsevier Science, Amsterdam, Netherlands, 1986.

[27] Jacques Cohen Constraint logic programming languages. Communications of the ACM, 33(7):52-68,1990.

1