Algebraic programming system APS
(user manual)

A.A. Letichevsky, J.V. Kapitonova, V.A. Volkov, A. Chugajenko, V. Chomenko, V. Peschanenko
Glushkov Institute of Cybernetics,
National Acad. of Sciences of Ukraine
Kiev, Ukraine
E-mail: let@iss.org.ua

May 22, 2008

Contents

1 Introduction

Algebraic programming is programming based on rewriting. It extends functional programming and has applications in solving computer algebra problems (word problem in finitely defined algebras, completion algorithms like Knuth-Bendix or Buchberger), and in operational semantics of programming languages (executable algebraic specifications of software components, definition of operational semantics of programming languages, developing interpreters and prototypes of software components).

In the difference from 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. The APS integrates four main programming paradigms in the following way. The main part of a program can be written in the form of rewriting systems. Imperative and functional programming are used for the definition of strategies. Logic paradigm is realized on a base of rewriting using built-in unification procedure. The text of a manual is based on publications [1, 2]

This introduction contains some examples of simple algebraic programs based on rewriting, functions, and procedures.

1.1 Rewriting rules

Let us begin with simple problem of functional programming — computing Fibonacci numbers. Well known recursive definition of n-th Fibonacci number is the following system of relations.

```
F(0) = 1 ,
F(1) = 1 ,
F(n) = F(n - 1) + F(n - 2)
```

This system may be considered as a (recursive) definition of a function F or a system of rewriting rules which can be used to compute F(n). To write this system in APLAN the following statement must be written first.

```
INCLUDE <gen_obj.ap>
```

This statement includes some standard definitions and especially provides the use of arithmetic operations ”+”, ”-”, signs ”=”, ”,”, and also some other syntactic notions defined in a module gen_obj.ap. Now the name of a system must be defined

```
NAME R;
```

and assigned a value by means of initial assignment (Example 1 in APS folder):

```
R:=rs(n)(
  F(0) = 1 ,
  F(1) = 1 ,
  F(n) = F(n - 1) + F(n - 2)
);
```

The first line of this assignment shows that the value of a name R is a system of rewriting rules (rs) and n is the only variable used in a system. General definition of syntax of rewriting rule systems is the following.

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

The system R can be applied for instance to the expression T = F(10) and this expression will be transformed in the following way:

F(10) = F(10-1)+F(10-2) = F(9)+F(8)

In APS this transformation is performed in a one step because the arithmetical operations are interpreted and are performed if it is possible at each rewriting step. The next step of rewriting can be performed in one of two ways dependently of what occurrence of F(n) the third rule of a system will be applied to. Let it be the first occurrence that is F(9). Then the new expression will be

T = (F(8)+F(7))+F(8)

If we shall continue in a similar way, that is select the leftmost (innermost) occurrence each time we shall obtain the sequence

```
T = ((F(7)+F(6))+F(7))+F(8) =
= (((F(6)+F(5))+F(6))+F(7))+F(8) =
.................................
= (...((F(1)+F(0))+F(1))+....)+F(8)
```

Now the third rule can be applied as well as the second one. Let us use the principle which is on a use in all main strategies of rewriting in APS. This principle is that in each case the rule to be applied is the first rule which can be applied in the current state of a computation. Therefore the next two steps will give us the following expressions:

```
T = (...((1+F(0))+F(1))+....)+F(8) =
= (...((1+1)+F(1))+...)+F(8)
```

In a new state there appeared one more possibility: to add 1+1 before the rewriting of F(1). The application of arithmetic operations to constants can be considered also as a rewriting with implicitly given rewriting rules (addition and multiplication tables). Therefore according to the leftmost strategy the addition must be performed first. The next steps of rewriting:

```
T = (...(2+F(1))+...)+F(8) =
= (...((2+1)+F(2))+...)+F(8) =
= (...(3+F(2))+...)+F(8) = ...= 55+34 = 89
```

Those fact that during rewriting only leftmost occurrence has been chosen for applying rewriting rules is not essential for this rewriting system. To obtain the needed result by means of the rule system R it is sufficient to use an arbitrary rewriting strategy (algorithm of rewriting) which satisfy the following conditions.

1. One of the rules of a system is applied or arithmetic operation is performed at each step of rewriting.
2. The choice of a rule is made according to the sequence in which rules has been written.
3. Rewriting continue till it is possible, that is till there are occurrences to which rules are applicable or arithmetic operations over numbers can be performed.

The strategy which satisfy these conditions is called *final* strategy. In APS one can use built-in strategies of rewriting or write his own strategies which meet the demand of a problem being solved in the best way. The call of a built-in strategy is an internal procedure call. Usually it has a form $s(T,R)$ where s is the name of a strategy, T — the name of an algebraic data structure to which a system is applied, R is a name of a system of rewriting rules.

There are two built-in final strategies of rewriting *applytb* (top-bottom iterative application) and nbt (bottom-up iterative application). Any of them may be used for computing n-th Fibonacci number. It is sufficient to add the definition of a name T, initial assignment to this name and a task which calls a strategy and prints the result to a file which contains the definition of R. The corresponding text can looks like the following.

```
NAME T;
T:=F(10);
```

```
task:=applytb(T,R),prn(T);
```

The name task is a standard name of a system. It is defined in the file std.ap and it is not necessary to define it once more. The statement prn(T) prints the value of a name T to a screen. To execute this task using command line

```
aps -i fib.ap
```

This command interprets a procedure task (this is default name of input point to a program, to change default input point you can use –t <some name> command) in this module.

Let us consider now how the strategies applytb and applybt work. First it is useful to understand how the algebraic expressions are represented as labeled trees. The nodes of such a tree corresponds to subexpressions of a given expression. Each node is labeled by the main operation of a subexpression if this subexpression is not a primary one, as for instance number or symbol. The nodes corresponding to the primary ones are labeled by these subexpressions. If a node corresponds to a subexpression $f(x_1, \dots, x_n)$, where f is an operation of arity n, then this node is connected by edges numbered by 1, . . . , n with the nodes corresponding to the subexpressions $x_1, \dots, x_n$.

A tree corresponding to the expression F(n • 1) + F(n • 2) can be represented by one of two ways depending on what is the meaning of the symbol F. It can be defined in APLAN as a unary operation by statement

```
MARK F(1);
```

In this case the symbol F labels two nodes of a tree. If the symbol F appears without definition it is considered as a symbolic atom and the main operation of an expression F(x) is the binary operation application which is denoted simply by concatenation of two expressions. The first argument of this operation in the expression F(x) is an atomF, the second argument is an expression x.

Both strategies *applytb* and *applybt* are based on the left depth first bypass of a tree. Each node is visited two times during this bypass: first time when the strategy moves top bottom, second time during the move bottom up. The strategy *applytb* (apply top-bottom) applies a rule system to an expression corresponding to the current node visiting this node from above. A strategy is applied to this node as many times as possible. The strategy *applybt* (apply bottom-up) does the same but only when visiting a current node from below. If during the complete bypass at least one rule has been applied the bypass is repeated and the strategy works until no one rule has been applied. Each strategy performs also all admissible simplifications including the execution of arithmetic operations when moving bottom-up. Therefore both strategies are final and each can be used to transform an expression F(n) where n is a natural number. The system R can be applied also to complex expressions containing the calls of function F for instance the expression F(F(n)). But in this case the strategies operate differently. The strategy *applybt* will compute the value of an expression while the strategy applytb will perform an infinite rewriting:

```
F(F(10)) = F(F(10)-1)+F(F(10)-2) =
(F((F(10)-1)-1)+F((F(10)-1)-2))+F(F(10)-2) = ...
```

One can avoid the infinite rewriting if use the conditional rewriting rules. It is worth while applying the third rule of a system R to an expression R(n) only if n is a nonnegative integer (Example 2). The corresponding condition is expressed in APLAN as

```
isint(n)&(n>0)
```

and can be added to the third rule. A new system can be presented in the following way (Example 3).

```
R:=rs(n)(
  F(0) = 1 ,
  F(1) = 1 ,
  isint(n)&(n>0)->(
    F(n) = F(n - 1) + F(n - 2)
  )
);
```

Now when the third rule is protected from undesirable applications the system R can be applied using arbitrary final strategy. If this system is applied to an arbitrary expression with occurrences of F then all subexpressions of a type F(n) where n is a nonnegative integer will be computed.

Let us now consider the rules for computing Fibonacci numbers from another point of view. It is easy to see that any final strategy must perform not less than exponential number of steps to compute F(n). Really after an application of the third rule the computation of F(n) is reduced to the computation of F(n•1) and F(n•2). These computations will be performed independently and computation of F(n • 1) is reduced to the computation of F(n • 2) and F(n • 3). Therefore F(n • 2) will be computed two times, F(n • 3) — three times and so on.

To understand how the computations can be improved let us consider once more the computation of F(10) allowing the application of arbitrary algebraic simplifications other then performing arithmetic operations over integers. We have

```
T = F(10) =
        (F(8)+F(7))+F(8) = 2*F(8)+F(7) = 2*(F(7)+F(6))+F(7) =
            3*F(7)+2*F(6) = 3*(F(6)+F(5))+2*F(6) = 5*F(6)+3*F(5) = ...
            ... = 89
```

It is easy to write general form of rules which are used here. They are

$$a * (x + y) + b * x = (a + b) * x + y$$

and special cases of this rule when $a = b = 1$ and when $b = 1$:

$$a * (x + y) + b * x = (a + b) * x + y,$$
$$a * (x + y) + x = (a + \mathbf{1}) * x + y,$$
$$(x + y) + x = 2 * x + y$$

If these rules are added to the system R the rewriting of F(n) can be done on the number of steps proportional to n. But only if a proper strategy will be used. Really the strategies *applytb* and *applybt* now do not work. Each of these strategies will apply the third rule before the new rules will be applied and the same waste will be done as without these rules. The strategy needed for us must work so that at each step of rewriting it is applied to the leftmost outermost occurrence of a term to which a system is applicable. This strategy exists among built-in strategies of APS. The name of this strategy is *lmt* (leftmost outermost, the famous strategy of lazy computation). A new system of rewriting rules is (Example 4)

and new task is:

```
task:=lmt(T,R),prn(T);
```

The strategy *lmt* defined in strat.ap.

One small disadvantages of this system is a large number of rules. This disadvantages can be avoid if more careful analysis of a general state of computation is applied. This state can be described by an expression

$$a * \mathrm{F}(n) + b * \mathrm{F}(n-1), (2)$$

which is obtained just after the fourth application of a system and is repeated after each two steps. Let us denote the function defined by this expression as f(a, b, n). It is easy to see that

if n > 0 then
F(n) = 1 * F(n) + 0 * F(n • 1) = f(1, 0, n),
f(a, b, n) = f(a + b, a, n • 1)

Therefore instead of computing of F one can compute a function f setting f(a, b, 0) = b and using the following rewriting system (Example 5)

```
R2:=rs(a,b,n)(
   f(a,b,0) = a,
   f(a,b,n) = f(a+b,a,n-1),
   F(n) = f( 1,0,n )
);
```

To control this system more simple strategy is needed. This strategy applies a system only to the expression itself without considering its subexpressions. The name of this strategy is *appls*. The computations would be faster if the symbol f is defined as an operation of arity 3, not as an atom or name. It is easy to see that the system R2 corresponds to the well known procedure program for computing F(n), but it is represented in algebraic form and has been obtained exclusively by algebraic methods without using any procedural reasoning. This is the main peculiarity of algebraic programming. It allows reducing procedural constructions to the necessary and simple minimum and express the mostly essential properties of algorithms in mathematical (algebraic) form.

1.2 Procedures

Imperative programs are sequences of statements separated by ”,” or ”;”. Imperative program can be the value of a name and can be called via system command such as

```
aps –i <name of a module>
```

The syntax of a statement is the following:

```
<statement> ::= <basic statement>| <conditional statement>
| <while statement> | <do statement> | <internal call>
| <external call> | return | return(<algebraic expression>)
| (<program>)
<basic statement> ::= <set statement> | <assignment statement>
<set statement> ::= <selector> --> <algebraic expression>
<assignment statement> ::= <name> := <algebraic expression>
<selector> ::= <name>
| arg(<selector>,<sequence of expressions separated by ",">)
<conditional statement> ::= <condition> -> <statement>
| <condition> -> <statement> else <statement>
<while statement> ::= while(<condition>, <statement>)
<do statement> ::= do(<name>)
<internal call> ::= <internal procedure name>
(<actual parameter list>)
<internal name> ::= <atom>
<external call> ::= <procedure definition or name of procedure
definition>(<actual parameter list>)
<actual parameter> ::= <algebraic expression>
```

Syntax of procedure definitions:

```
<procedure definition> ::= proc(<formal parameters list>)
<local names> <statement>
```

<local names> ::= loc(<local names list>) | <empty>
<formal parameter> ::= <identifier>

Example of a program:

```
task:=(
  prn T;
  yes:=1;
  while(yes,
     applr(T,R)
  );
  prn T
);
```

This is a typical structure of a program with the main part written as a system of rewriting rules R. Statement applr(T,R) are internal procedure calls. They call internal procedures written in C (low level language of APS). applr is one of the basic strategies of rewriting. It applies the system R to the root of a term T one time. If the application is successful, the name yes gets value 1, otherwise it gets the value 0.

Another example is the procedure definition of a strategy *applytb*.

```
NAMES applytb,appl_tb_rec;

applytb:=proc(t,R)loc(Y)(
   dowhile(Y:=appl_tb_rec(t,R),Y)
);

appl_tb_rec:=proc(t,R)loc(Y,s,i)(
   appls(t,R);
   Y:=yes;
   forall(s=arg(t,i),
      Y:=appl_tb_rec(s,R)|/Y
   );
   t:=can(t);
   return Y
);
```

Second procedure definition is also used as a function because it returns a value Y. forall(s=arg(t,i),P) is an abbreviation for a loop on all immediate subterms s of a term t and can be expressed using while statement. You can use the appl_tb_rec (appl_bt_rec) strategy separetly by name ntb(nbt). The introducing of such abbreviations will be discussed later.

1.3 Functions

There are the following two ways to introduce functions: by means of rewriting rule systems and by means of (external or internal) procedures which return values. They can be called explicitly or by means of the names of rewriting rule systems procedure definitions, or the names of internal procedures in algebraic expressions.

The following example is a program which prints the table of Fibinacci numbers with stop (the internal procedure wait ent()) after each 10 lines. It must be mentioned that internal procedures can be used as functions which return the value 1 (Example 6).

```
NAMES tab,turn_right,prtab;

tab:=rs(x,y,z,u,n,p)(
   ( (x,y,z), 0) = (F(0)=1),
   ( (x,y,z), 1) = (F(0)=1,F(1)=1),
   ( (x,y,z), 2) = turn_right(((z,y),x)),
   ( (F(n)=x,p=y,z), u) = tab((F(n+1)=x+y,F(n)=x,z,p=y), u-1)
);

turn_right:=rs(x,y,z)(
   ((x,y),z) = turn_right(x,y,z)
);

prtab:=rs(x,y,n,m)(
   (n>0)&(n mod 10 == 0)->(
      (F(n)=m,y) = prn(F(n)=m)&wait_ent()&y
   ),
   (x,y) = prn(x)&y,
   (x=y) = prn(x=y)&wait_ent()
);

task:=(
   T-->tab((F(2)=2,F(1)=1,F(0)=1),T),
   appls(T,prtab);
   wait_ent()
);
```

2 Structure of APS

2.1 Data structures

The main data type in the system is the algebra $T_{\sim}(Z)$ of terms (trees) generated by the set Z of primary objects and the operations of the signature ˜. This algebra is considered as absolutely free ˜-algebra and it is extended to the algebra $T_\Omega^*(Z)$ of infinite (but finitely represented or rational) trees. As a values of names these structures may have common parts and may be used to represent arbitrary labelled graphs (graph terms). This possibility is realized on imperative level and usually is ignored on the level of algebraic programming.

2.2 System objects

There are three types of system objects: *algebraic programs* (ap-modules), *algebraic modules*(a-modules) and *interpreters*.

Algebraic programs are texts in APLAN language. Each program contains the description of signature ˜ with syntax for constructing algebraic expressions (terms). It defines also the set of names X and atoms A. These objects together with numbers and strings constitute the set Z of *primary objects*. Three sets mentioned above define the type (˜,X,A) of apmodule. The types of ap-modules are partially ordered by the inclusion relation (symbols of ˜ are considered jointly with their descriptions which includes in particular the arity of each symbol). If (˜,X,A) ⊂ (˜•,X•,A•) ap-module M of the type (˜•,X•,A•) is said to belong to the class C(˜,X,A). Two classes are said to be compatible if they have common lower bound that is a common subclass. Parameters of this subclass contains parameters of both compatible classes. Algebraic program defines also the initial values of names. These values are objects of the type T˜(Z)

Algebraic modules contain internal representation of the data structures defined in apmodules. They are being created by system commands that refer to ap-modules as a new object generators. Algebraic module M generated by program P inherits its type and initial values of names. The notion of a-module is dynamical one. It has a state which may changes in time. The change of the state of a-module takes place as a result of executing procedures located in it by means of interpreters. The ordering on the set of types of a-modules as well as the notion of classes CA(˜,X,A) for them are defined similarly to the corresponding notions for ap-modules. Thus the ap-modules plays the same role w.r.t. a-modules as the classes w.r.t. the objects in the object-oriented programming.

2.3 States

The state of a-module of the type (˜,X,A) consists of two components. First is the memory state that is the mapping $\sigma: X \rightarrow T = T_\Omega^*(Z)$. The second component is an equivalence relation on the set of nodes of ˜-trees which are the values of names. Two nodes can be identified by this equivalence only if they are labeled with the same operations and subterms generated by these nodes are isomorphic. And if two nodes are identified then the corresponding sons (the arguments of an operation which labels nodes) must be identified also.

2.4 System interpreters

They are programs destined for the interpretation of the procedures written in APLAN. They are developing in C language on the base of libraries of functions and data structures to work with internal representation of system data structures. Each interpreter is connected with the distinct type (˜,X,A) which defines classes CI(˜,X,A) the interpreter belongs to in the similar way as for modules. This type defines the restriction to algebraic modules which can be executed by the given interpreter. All of them must belong to the class which is compatible with the class CA(˜,X,A).

Each interpreter specifies the operational semantics of APLAN for the given class of amodules and provides efficient implementation of the procedures, functions and strategies of rewriting for the systems located in the given module. Classification of the interpreters given above is syntactical one and there exists more detailed classification w.r.t. to their semantical properties. System commands provide the possibility to make up the following actions:

- Creating of a new a-module x by means of a program y: “create x y”, x and y are the names of files, y already exists, x is to be created as a new file.
- Completion of a-module x with a program y: “complete x y”, x and y are the names of already existing files.
- Executing program x of an algebraic module y by means of interpreter z: “z x y”, z is the name of executable file, x is the name defined in a-module y.

System commands may be executed by the requirement of a user or used as internal calls in algebraic programs. Such calls along with some additional possibilities provides the interaction among algebraic modules.

3 APLAN

3.1 Generating ap-modulus

General syntax used for preparing ap-modules is a syntax of algebraic data structures and it does not distinguishes between special data structure such as rewriting rules, procedures and other objects. Algebraic program is defined as the sequence of sentences. There are the following types of sentences:

- name descriptions,
- mark descriptions,
- initial assignments,
- inclusions,
- comments.

```
<name description> ::= NAMES <sequence of names separated by "," >;
<name> ::= <identifier>

<mark description> ::= MARK <sequence of mark descriptions
elements separated by "," >;
<mark description element> ::=<mark symbol>(<arity>)
```

| <mark symbol>(2, <priority>, "<infix notation>")
<mark symbol> ::= <identifier>
<arity> ::= <positive integer> | UNDEF
<priority> ::= <positive integer>
<infix notation> ::= <sequence of signs>

<initial assignment> ::= <name> := <algebraic expression>;
<algebraic expression>::=<primary expression>|<prefix expression>
|<application>|<infix expression>
<primary expression>::=<integer or rational number> | <string>
| <empty object> | <name> | <atom> | VAL <name>
|(<algebraic expression>)
<empty object>::= ()
<application>::=<algebraic expression> <algebraic expression>
<prefix expression>::=<mark symbol>(<sequence of algebraic expressions separated by ",")
<infix expression>::=<algebraic expression><infix notation>
<algebraic expression>

In the prefix expression $\omega(x_1,\dots,x_n)$ where • is a mark symbol, the number of arguments must be equal to the arity of this mark if arity is integer and may be arbitrary if arity = UNDEF. The priority of infix expression x•y where • is infix notation is defined as the priority of •. Expression x must be primary expression, or application, or infix expression with priority larger than priority of • and if y is infix expression its priority must be larger or equal to the priority of •.

<inclusion> ::=INCLUDE <file name inserted into "<>">
| INCLUDE "<file name >"

Comments are pointed out by brackets /* */. Strings are symbol sequences inserted into "".

Inclusion sentence INCLUDE x means that text of module x should be inserted instead of the sentence. When the name description is processed new names mentioned in it are added to the set of names. Mark descriptions extend the signature ˜ of a-module. Besides the algebraic operation themselves marks may be used as function or predicate symbols, names of types, constructors of data structures and so on. This explains why the neutral term mark is used instead of operation or function. When infix notation is presented in mark description it may be used for infix representation of expressions. In this case priority helps to omit brackets. When the arity is UNDEF, the mark may be used with different arity > 0 (this mark may be associated with infinite family of operations). The only marks that initially exist and may be used without descriptions are binary application with empty infix notation and mark ARRAY(UNDEF) which may be used for array construction. Application always has the highest priority in the system. Atoms are identifiers which occur in program and were not described as names or marks or infix notations.

Internal representation of algebraic expressions are ˜-trees constructed in obvious way. After processing of name description the value of each name is initialized by empty object which is only one that exists before initialization. Initial assignment x:=y makes the value of x equal to the term represented by algebraic expression y. When this object is created the values of names are not substituted except of the case when the name z follows the symbol VAL. In this case value of z will be referenced instead of VAL z. The use of this tool makes it possible to identify the nodes of internal representation of trees. If VALz = arg(y, p) then after this assignment the equivalence arg(x, p) = arg(z, p)(•) will appeared.

When the name or mark is redefined the previous definition is canceled. The same is true about initial assignment. It means that it is impossible to create objects with loops. Indeed, after initial assignment x:=. . .VAL x . . . all occurrences of VAL x shall be replaced by empty object even if x was already initialized.

3.2 Imperative programs

The syntax of programs and external procedures has been described in the section 1.2. To explain more precisely the evaluating of expressions, semantics of basic statements and transferring parameters, formal model for representation of module states must be introduced.

Representation of states. Let us consider the state (˜, •) for the module of the type

(•,X,A). As it was defined above ˜ is the mapping from X to $T^*_\Omega(Z)$ and • is the equivalence relation on the set of nodes of rational trees from $T^*_\Omega(Z)$. Each node is uniquely represented by a pair (x, p), called X-occurrence, where $x \in X, p - (i_1, \dots, i_m)$ is a sequence of positive integers represented the path in a tree (or occurrence in a term) ˜(x). This path is restricted in an obvious way by arities of intermediate nodes. In APLAN this node is also a root node of a subterm represented by an expression arg(t, $(i_1, \dots, i_m)$), where arg is a standard binary operation in •, t = ˜(x).

Let U = $\{u_1, \dots, u_m\}$ be an alphabet of symbols set to one-to-one correspondence with

the classes of •. The symbols of U will be identified with corresponding classes and we shall write (x, p) $\in$ u to claim that (x, p) is in the class corresponding to u. They also will be considered as nodes of a graph representing the set of data structures contained in the given module in the current state.

Let (x, p) $\in$ u, •(x) = t, arg(t, p) = ˜$(t_1, \dots, t_n)$. Then if (x, (p, i)) $\in v_i$, i = 1, . . . , n, we shall write u • ˜(v1, . . . , vn) and call this expression the decomposition of the node u. If arg(t, p) $\in$ Z the decomposition of u is u • arg(t, p). Decomposition of class u does not depend on the representative (x, p) of this class and is determined uniquely by the class itself. If (x, ())$\in$ u then x • u will be called the decomposition of the name x. The set of decompositions for all the nodes and names is called node representation of the state (˜, •). It may be easily shown that node representation uniquely defines the state it represents and for any state its node representation is defined up to the renaming of nodes.

It is convenient to extend the notion of representation allowing the rhss (right hand sides) of decompositions to be the arbitrary finite terms over T.(Z ∪ U). Using this extension one may eliminate some wasteful nodes. The node u is called to be wasteful if it occurs in the rhss of decompositions no more than once. The wasteful node u may be eliminated after replacing of its unique occurrence by rhs of its decomposition. The representation that has no wasteful nodes is called minimal in the difference of the representation defined above which is called maximal.

Computing values. There are two kinds of values that may be computed for algebraic expression t. The first kind denoted as val(t) belongs to the set T.(Z ∪ U) and is expressed by means of minimal representation of current state. Another is denoted as Val(t) and belongs to the set $T^*_\Omega(Z)$. The dependency between two kinds of values is expressed by the formula:

$$\mathrm{Val}(t) = (\mathrm{val}(t))\bullet^{\tilde{}}$$

where $\bullet^{\tilde{}}$ is the limit of • k, and • is a substitution of rhss of node and name decompositions instead of nodes and names in the rhss of all of the node and name decompositions. The second kind of value does not depend on equivalence • and is used in "invariant" reasoning about algebraic programs. The first kind is used to define precisely the operational semantics of procedure tools. Function val(t) substitutes the values of names and reduces the expression to the main canonical form using interpreters of operations (functions) ˜. Formal definition includes the following rules:

isname(x), x • t ∈ r ⟹ val(x) = t;
isfun(f) ⟹ val(f(x)) = $_{\mathrm{appl}}$(nd(f), val(x));
val(‘(t)) = t;
val(˜(t$_1$, . . . , t$_n$)) = $_{\sim}$(val(t$_1$), . . . , val(t$_n$));
isname(x), x • t ∈ r ⟹ nd(x) = nd(t);
nd(x) = x

The rules depend on a current state of a module represented by node decomposition r. Each rule may be applied only if previous one is not applied. Expression isfun(f) is true if F = nd(f) is procedure definition or rewriting rule system. If F is a procedure definition, this procedure is executed accepting an expression x as a list of actual parameters and the value is the value returned by this procedure. If F is a rewriting rule system this system is applied to x with the strategy applr.

Executing procedures can change current state of a module by side effects. So the order of computing the values of arguments of an operation ˜ is essential.

Basic statements. In both cases (set and assignment) the value s $\in$ T.(Z $\cup$ U) of rhs is computed. Consider set statement. If lhs is a name x its decomposition is replaced by x • s. Let us consider the statement arg(x, p)•• >t. The value of p must be the sequence (i_1, . . . , i_n) of positive integers. If (x, i_1, . . . , i_n•1)   u and u • q $\in$ r then s is substituted instead of the in-th argument of q. Of course the arity of q must be more or equal to in.

Assignment x:=t acts differently. First if s is a node the rhs of the decomposition for this node is taken instead of s. If the rhs of the decomposition for x is not a node then assignment is equivalent to set statement. Otherwise if x • u, u • q $\in$ r decomposition u • q is changed to u • s.

External calls. Formal parameters and local names temporarily added to module as names. Formal parameters are assigned the values of actual parameters and the body of procedure is executed. After that formal parameters and local names are deleted from module. Return statement produces the value which is used when the procedure is occurred in algebraic expression.

Internal calls. Address to procedures implemented on the level of C language. The number of formal parameters and how to transfer them (compute values or not) defined accordingly to specifications of internal procedure.

Executing programs. The process of executing programs is a traditional one. Statements are executed one after another. Conditional and while statements have usual meaning. The only interesting moment is the use of extension of an imperative APLAN. If an interpreter of a program meets the unknown basic statement, it tries to compile it by means of a rewriting system compile which is the standard tool of APS. A system compile can be changed and extended by user. A typical content of this system is the following (gen_obj.ap).

```
NAMES _p,_sp,_ptr;

_p :=nil;
_sp:=nil;

MARK case(2);

compile:= rs(x,y,z,u,i)(
  define x = ‘(defcall ‘(x)),
  (arg(arg(x,y),z)-->u) = compile(arg(x,conc(y,z))-->u),
  (arg(x,y)--> z ) = ‘(set(x,y,z)),
  ( x(i)--> z ) = ‘(set(x,i,z)),
  ( (x.y)--> z ) = ‘(set_comp(x,y,z)),
  ( x --> z.u) = set_obj_comp ‘(x-->z.u),
  ( x --> z ) = ‘(setname(x,z)),
  dowhile(x,y) = (x, while(y,x)),
  dowhile x y = (x, while(y,x)),
  for(x,y,z,u) = (x, while(y,(u,z))),
  loop(x) = while(1,x),
  forall(x=arg(y,i),z) =
  for(i-->1,i<= ‘(ART(y)),i:=i+1, x-->arg(y,i); z ),
  forallw(x=arg(y,i),u,z) =
    for(i-->1,(i<= ‘(ART(y))) & u,i:=i+1,
       x-->arg(y,i); z
     ),
  forall(x in list y, z) =
    (_sp-->(_p,_sp);
   _p-->y;
    while( ‘(is_type( _p,((),()) ) ),
       x -->arg(_p,1);
      _p-->arg(_p,2);
      z
    );
    x-->_p;
   _p -->arg(_sp,1);
   _sp-->arg(_sp,2);
   z
  ),
  forall(x in set y, z) =
    (_sp-->(_p,_sp);
   _p-->y;
   while(‘(is_type( _p,((),()) ) ),
     x -->arg(_p,1);
    _p-->arg(_p,2);
    z
   );
   _p -->arg(_sp,1);
   _sp-->arg(_sp,2)
  ),
forallw(x in list y, u, z) =
  (_sp-->(_p,_sp);
    _p-->y;
    while(‘(is_type( _p,((),()) )&u ),
      x -->arg(_p,1);
     _p-->arg(_p,2);
     z
    );
    x-->_p;
   _p -->arg(_sp,1);
   _sp-->arg(_sp,2);
   u->z
  ),
  forallw(x in set y, u, z) =
    (_sp-->(_p,_sp);
   _p-->y;
   while(‘(is_type( _p,((),()) )&u ),
     x -->arg(_p,1);
     _p-->arg(_p,2);
     z
   );
   _p -->arg(_sp,1);
   _sp-->arg(_sp,2)
  ),
  case(x,y->z;u)= starg(x,2,y)->z else case(x,u),
  case(x,y->z )= starg(x,2,y)->z,
  case(x, z )= z,
  branch(y->z;u)= y->z else branch(u),
  branch(y->z )= y->z,
  branch( z )= z
);
```

4 Strategies

4.1 Basic strategies

Strategies of rewriting in APS are based on two main internal 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 simple rule is usual: matching lhs (left hand side) with t, if success then substitution of variables to lhs and replacement of t by rhs. Before replacing the redex rhs is reduced to main canonical form by the rules similar to computing values but without substituting the values of names.

To be more precise let us consider the statement applr(x,y) and let t = val(x),R = val(y). Let z be an auxiliary name with the decomposition z • t, $x_1, . . . , x_n$ be the variables of a system R, l = r be the rule such that its lhs l is matched with t and $u_1, . . . , u_n$ are zoccurrences corresponding to the values of variables $x_1, . . . , x_n$ (nodes of some representation of current state). If the rule is not left linear that is l has more than one occurrence of some variable the first occurrence of this variable is considered. Substitution of rhs is then equivalent to the assignment z:=CAN(r), the function CAN being defined by the following rules:

$CAN(x_i) = u_i$;
$isfun(f) \Longrightarrow CAN(f(x)) = {}_{appl}(nd(f), x)$;

$\mathrm{CAN}(`(s)) = s[x_1 \bullet u_1, \ldots, x_n \bullet u_n];$
$x \in Z \Longrightarrow \mathrm{CAN}(z) = z;$
$\mathrm{CAN}(\tilde{}\,(t_1, \ldots, t_n)) = \quad \tilde{}\,(\mathrm{CAN}(t_1), \ldots, \mathrm{CAN}(t_n));$

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 computing the function CAN. If the result is 1, applying the rule continue as usual. Otherwise application is canceled.

The statement appls(t,R) calls applr(t,R) while yes = 1. Note that if the name of rewriting system is occurred in the right hand side of a system, then a system is applied recursively. This is why the strategy, realised by procedure applr is called a recursive strategy of rewriting. The strategy, realised by appls is called the iterative one.

4.2 Canonical forms

For algebraic computations it is typical to consider algebraic expressions up to some congruence consistent with the identities of the algebra that defines subject domain. In APS this idea is realised by means of a function CAN. This function defines the equivalence $t = t\bullet(\mathrm{CAN}) \Longleftrightarrow \mathrm{CAN}(t) = \mathrm{CAN}(t\bullet)$ which must be the congruence: $t_1 = t\bullet_1(\mathrm{CAN}), \ldots, t_n = t\bullet_n(\mathrm{CAN}) \Longrightarrow \tilde{}\,(t_1, \ldots, t_n) = \tilde{}\,(t\bullet_1, \ldots, t\bullet_n)(\mathrm{CAN})$. This is equivalent to the existence of function can such that

$\mathrm{CAN}(\tilde{}\,(t_1, \ldots, t_n)) = \mathrm{can}(\tilde{}\,(\mathrm{CAN}(t_1), \ldots, \mathrm{CAN}(t_n))).$

Function CAN defined above does not define the congruence because application and quote operations prevent it. But if the latter operations are ignored it does. Really, function can is defined by means of operation interpreters:

$$\mathrm{can}(\tilde{}\,(t_1, \ldots, t_n)) = \quad \tilde{}\,(\tilde{}\,(t_1, \ldots, t_n))$$

Another important property of CAN is that it must define the canonical form for given congruence: $t = \mathrm{CAN}(t)(\mathrm{CAN})$. This property is equivalent to idempotency of CAN: $\mathrm{CAN}(\mathrm{CAN}(t)) = \mathrm{CAN}(t)$. When CAN possesses idempotency it is called to be correct.

Term *t* is called to be normalized w.r.t. can if can(s) = s for any subterm s of term t. The following condition is sufficient for the correctness of CAN: if $t_1, \ldots, t_n$ are normalized w.r.t. can then $\mathrm{can}(\tilde{}\,(t_1, \ldots, t_n))$ is also normalized.

This condition reduces the check for correctness of CAN to the analysis of normalization properties of operation interpreters.

As a simple example realized in the most of APS system interpreters it may be mentioned operation interpreters that evaluate constant computations for arithmetic and boolean operations implementing some simple identities such as $x + 0 = x$ or $x|/1 = 1$.

The mostly often used operations are defined in the file nstd.ap. Below is the content of this file with some comments for interpretation and use of operations (marks).

File std.ap used in some examples below is the same but disjunction id denoted as ||, not as |/ (std.ap).

```
MARKS
/* Arithmetical and algebraic operations and functions */

POW( 2, 60, "^"), /* power x ^ y */
MOD( 2, 59,"mod"), /* residual x mod y */
Mult( 2, 58, "*"), /* multiplication x * y */
DIV( 2, 57, "/"), /* division x / y */
MLT( 2, 56, "$"), /* uninterpreted */
SUB( 2, 55, "-"), /* subtraction x - y */
ADD( 2, 54, "+"), /* addition x + y */

/* Predicates */

LE ( 2, 40, "<="), /* less or equal x <= y */
LS ( 2, 40, "<"), /* less x < y */
ME ( 2, 40, ">="), /* more or equal x >= y */
MR ( 2, 40, ">"), /* more x > y */
EQ ( 2, 11, "=="), /* equality of numbers x == y */
EQU ( 2, 11, "=" ), /* uninterpreted */

/* Logical connections */

~ ( 1, 30), /* logical negation ~(x) */
AND( 2, 29, "&"), /* logical and x & y */
OR ( 2, 28, "|/"), /* logical or x |/ y */
IFF( 2, 27, "<=>"), /* logical equivalence x <=> y */

/* L2B operations */

SET ( 2, 20, "-->" ), /* set statement x-->y */
ASS ( 2, 20, ":=" ), /* assignement statement x:=y */
ELSE ( 2, 19, "else"), /* used in conditional statement:
x -> y else z */
IF ( 2, 18, "->" ), /* implication x -> y and separator
in conditional statements */
do (1), /* do statement: do(p) */
while(2), /* while statement: while(x,y) */

/* Separators */
```

```
comma( 2, 7, ","),
LL ( 2, 5, ";"),

/* Special functions */

‘(1), /* ‘(x) = x */
arg (2), /* arg(f(x1,...,xn)) = xi */
ART (1), /* ART(x) = arity of x */
CAN (1), /* Basic canonical form */
IFTH(3), /* IFTH(x,y,z) = if x then y else z */
vl (1), /* vl(x) = copy of value of name x */
VL (1), /* VL(x) = substitution of values to term x */
intr(2),
copy(1), /* copy(x) is copy of x if x has no loops */
mrg (1);
NAMES indprn,indprog,indpl,
indts,indtm,task,nil,
extcan;
NAMES in,out,stack,proc_atom,rs_atom;
in :=0;
out :=0;
stack:=0;
nil:=nil;
extcan:=nil;
```

4.3 ac-operations

There are two kinds of associative and commutative operations that may be introduced in APS: arithmetical and boolean like ac-operations.

Arithmetical ac-operation ˜ is introduced jointly with operation of iteration   and two optional constants: neutral element e and annulator a. Except of associativity and commutativity the following identities are true:

$$
\begin{aligned}
&(x \;\; y)\tilde{}(x \;\; z) = x \;\; (y + z);\\
&x\tilde{}e = x;\\
&x\tilde{}a = a;\\
&x \;\; 0 = e;\\
&x \;\; 1 = x;\\
&e \;\; x = e;\\
&a \;\; x = a
\end{aligned}
$$

For boolean like operations the unary negation operation • is used and except of neutral element and annulator the outermost element o may be introduced. The identities for boolean like operations are:

```
x˜x = x;
x˜e = x;
x˜a = a;
•(•(x)) = x;
x˜•(x) = o
```

Information about ac-operations is collected in the data structure which is the value of standard name ac list. This structure is an array of ac-descriptions. Each description is 5-tuple. For arithmetical ac-operations the description is (()˜(), () (), e, a, nil). For boolean like operation the description is (()˜(), •(), e, a, o). If one of three constants is not used the symbol nil must be set in the corresponding position. As an example let us consider the following description:

```
ac_list:= ARRAY(
   (()+ (), ()$ (), 0, nil, nil),
   (()* (), ()^ (), 1, 0, nil),
   (()& (), ~(()), 1, 0, 0),
   (()|/(), ~(()), 0, 1, 1),
   (()><(), ()>^(), Delta, nil, nil),
   (()||(), ()|^(), Delta, 0, nil)
);
```

The first two operations are arithmetical ones. They are interpreted in the usual way. The next two are boolean also interpreted in the usual way (in the early written algebraic programs disjunction was denoted as ||, this is the case in the sections below). The last two operations are combination and parallel composition used for the implementation of the algebra of concurrent processes.

Ac-operations are supported by function mrg and two internal procedures merge and ord. These procedures and function are used to reduce expressions containing ac-operations to ac-canonical form that provides ordering and reducing similar members for arithmetical operations and simplifications for both types of ac-operations. Function mrg and procedure merge reduce to canonical form expressions of the type x˜y where x and y are already in canonical form.

4.4 Built-in strategies

The strategies described in this section are specified in APLAN by means of simple recursive procedures and are implemented on the level of C as internal procedures. All strategies can be applied only to dags (graph terms which are acyclic directed graphs) (strat.ap).

```
ntb:=proc(t,R)loc(s,i)( /* top-bottom */
  appls(t,R);
  forall(s=arg(t,i),
   ntb(s,R)
  );
  t:=can(t)
);

nbt:=proc(t,R)loc(s,i)( /* bottom-up */
  forall(s=arg(t,i),
   nbt(s,R)
  );
  appls(t,R);
  t:=can(t)
);
```

These two strategies are one path top-bottom and bottom-up strategies. They correspond to the simplest cases of call-by-name and call-by-value computations, correspondingly. The result of application of ntb is always a term in basic canonical form.

```
applytb:=proc(t,R)loc(Y)(
 dowhile(Y:=appl_tb_rec(t,R),Y)
);

appl_tb_rec:=proc(t,R)loc(Y,s,i)(
 appls(t,R);
 Y:=yes;
 forall(s=arg(t,i),
  Y:=appl_tb_rec(s,R)|/Y
 );
 t:=can(t);
 return Y
);

applybt:=proc(t,R)loc(Y)(
 dowhile(Y,Y:=appl_bt_rec(t,R))Y
);

appl_bt_rec:=proc(t,R)loc(Y,s,i)(
 Y:=0;
 forall(s=arg(t,i),
  Y:=appl_bt_rec(s,R)|/Y
 );
 appls(t,R);
 Y:=yes|/Y;
 t:=can(t);
 return Y
);
```

These two strategies are final that is the resulting term is always normalised if it is defined (if the strategy terminates).

```
ntr:=proc(t,R)loc(s,i)(
 yes:=1;
 while(yes,
   t:=can(t);
   appls(t,R);
   yes:=0;
   forallw(s=arg(t,i), ~(yes),
     ntr(s,R)
   )
 );
 t:=can(t);
 appls(t,R)
);
```

This strategy in some cases is more efficient than others.

```
lmt:=proc(t,R)loc(Yes)(
 dowhile(Yes:=lmt_rec(t,R),Yes)
);

lmt_rec:=proc(t,R)loc(Yes,s,i)(
 t:=can(t);
 appls(t,R);
 yes->return(1);
 forall(s=arg(t,i),
   Yes:=lmt_rec(s,R);
   Yes->return(1)
 );
 t:=can(t);
 return(0)
);
```

Leftmost outermost strategy is a famous strategy, which is complete for lambda calculus.

```
ntb2:=proc(t,R)(
  appls(t,R);
  (ART(t)>0)->ntb2 (arg(t,1),R);
  t:=can(t);
  (ART(t)>0)->ntb2 (t,R)
);

can_right:=proc(x,R)(
  appls(x,R);
  while(yes,
    (ART(x)>1)->can_right(arg(x,2),R);
    appls(x,R)
  )
);
```

These two strategies are used for the manipulation with lists, especially for computation on the branches of a search tree.

```
NAME ntb0;
ntb0:=proc(t,R)loc(s,i)(
  applr(t,R);
  forall(s=arg(t,i),
    ntb0(s,R)
   )
);

ntb0s:=proc(t,R)loc(s,i)(
  appls(t,R);
  forall(s=arg(t,i),
    ntb0(s,R)
  )
);
```

These two strategies are used when canonization is not needed. This is the case when solving language manipulation problems.

```
can_ord:=proc(t,R1,R2)loc(s,i)( /* top-bottom R1, bottom-up R2 */
  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)
);
```

This is one of the most important efficient strategy for reductions with ac-operations. It applies two systems of rewriting rules. The first one is applied when the strategy is moving top-bottom, the second one when it is moving bottom-up. One of the disadvantages of this strategy (for some cases) is that it can be not final. The next strategies are final for more wide class of terms in comparison with can ord. They use an auxiliary data structure, the name rpar which gets as a value the right hand side of a rule which was applied in the strategy applr.

```
NAME rpat;
can_ord2:=proc(t,R,S)loc(i,ar)(
  t:=can(t);
  appls(t,R);
  ar=ART(t);
  for (i:=1,i<=ar,i:=i+1,
    can_ord2(arg(t,i),R,S)
  );
  can_up2(t,S,0)
);

can_up2:=proc(t,S,pat)loc(s,i,ar,ar0)(
  dowhile(
  ~(equ(pat,0))->(
    ar0 = ART(pat);
    ar = ART(t);
    (ar0==0)->return 0;
    for(i:=1, i<=ar, i:=i+1,
      (i<=ar0)->s-->i else s-->ar0;
      can_up2(arg(t,i),S,arg(pat,s))
    )
  );
  applr(t,S);
  yes->pat-->rpat
 )~(yes);
  t:=can(t);
  merge(t)
);

can_ord3:=proc(t,R,S)loc(i,ar)(
  t:=can(t);
  appls(t,R);
  ar=ART(t);
  for (i:=1,i<=ar,i:=i+1,
    can_ord3(arg(t,i),R,S)
  );
  can_up3(t,S,0)
);

can_up3:=proc(t,S,dep)loc(i,ar)(
  dowhile(
    (dep>1)->(
      ar = cl_aryty(t);
        for(i = 1; i <= ar; i++,
          can_up3(cl_arg(t,i), S, dep-1);
    );
   t:=can(t);
   merge(t);
   applr(t,S);
   yes->dep:=depth(rpat)
  )~(yes)
);

depth:=proc(t)loc(i,ar,m,n)(
  m=0;
  ar=ART(t);
  for(i:=1,i<=ar,i:=i+1,
     n:=depth(arg(t,i))+1;
     (n>m)->m:=n
  );
  return(m)
);
```

```
can_atr:=proc(t,R1,R2,m)loc(s,i)( /* top-bottom R1, bottom-up R2 */
  appls(t,R1);
  forall(s=arg(t,i),
    can_atr(s,R1,R2,m)
  );
  can_atr_up(t,R2,m,0)
);

can_atr_up:=proc(t,R,m,n)loc(s,i)(
  (n>0)->(
    forall(s=arg(t,i),
      can_atr_up(s,R,m,n-1)
    )
  );
  appls(t,R);
  (yes)->(
    forall(s=arg(t,i),
      can_atr_up(s,R,m,m)
    )
  )
);
```

Strategy can atr is similar to attribute grammars.

```
lisp:=proc(t,R)loc(ar,i)(
  dowhile(
    ar=ART(t);
    (ar>0)->for(i:=1,i<=ar,i:=i+1,
      lisp(arg(t,i),R);
      t:=can(t)
    );
    applr(t,R)
  )yes
);
```

Lisp-like strategy.

```
stb:=proc(t,R)loc(ar)(
  t:=can(t);
  applr(t,R);
  yes->(
    ar=ART(t);
```

```
      for (i:=1,i<=ar,i:=i+1,
        stb(arg(t,i),R)
      )
    )
  );
```

The same as *ntb* but *applr* is used instead of *appls*. The specifications of built-in strategies can be used for the developing new ones by transformations and adding new properties. They are also useful for proving properties of algebraic programs (usually by means of structure induction). Some of thouse strategies are anbsant in the start.ap. So, you can define it manualy (don't foget define some names).

5 Algebra of logic

Following is a complete example of algebraic program written in the file log.ap. This program contains some tools to process propositional formulas (Example 7).

```
INCLUDE <std.ap>
INCLUDE <gen_obj.ap>

MARK subs(2);

/* Rules for eliminating <=>, ->, and de Morgan rules*/

NAME R;

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 */

NAMES R1,Q1;

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
);

/* Rules for DNF */

NAMES DR1,DQ1;

DR1 :=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
);

DQ1 :=rs(x,y,z)(
  (x || y)& z = (x & z) || (y & z) ,
  x &(y || z) = (x & y) || (x & z),
  (x & y)& z = x & y & z
);

NAME cnf;

cnf:=proc(x)(
  x-->copy(x),
  ntb(x,R),
  can_ord(x,R1,Q1),
  return(x)
);

NAME dnf;
```

```
dnf:=proc(x)(
  x-->copy(x),
  ntb(x,R),
  can_ord(x,DR1,DQ1),
  return(x)
);

NAME deM;

deM :=rs(x,y)(
  ~( ~(x) ) = x,
  ~(x || y) = deM(~(x)) & deM( ~(y)),
  ~(x & y) = deM(~(x)) || deM(~(y))
);

/* Rules for proving identities in logic */

NAME I1;

I1:=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
);

/*
--------------------------------------------------------------------
Strategy ntb2
NAME ntb2;
ntb2:=proc(t,R)loc(s,i)(
appls(t,R);
(ART(t)>0)->ntb2 (arg(t,1),R);
t:=can(t);
(ART(t)>0)->ntb2 (t,R)
);
--------------------------------------------------------------------
```

```
*/

                    NAME is_id;

                    is_id:=proc(x)(
                      x-->copy(x);
                      ntb(x,R);
                      x-->(1->x);
                      ntb2(x,I1);
                      return(x)
                    );

/* printing list constructed by means of binary operation y */

                    NAME prn_list;

                    prn_list:=proc(x,y)loc(z,n)(
                      line();
                      z-->x;
                      n:=0;
                      while(equ(type(z),y),
                        n:=(n+1) mod 10;
                        print(arg(z,1));
                        line();
                        (n==0)->wait_ent();
                        z-->arg(z,2)
                      );
                      print(z);
                      put("\nend of list")
                    );
```

The file gen_obj.ap contains some additional to std.ap definitions of marks, ac list and compile. Function subs (interpreted binary operation with infix notation) defines substitutions: subs((x=a,y=b,...),z) substitutes a,b,... instead of primary objects x,y,... to z. Procedure prn list is used for step-by-step printing of large lists (for example the list of disjuncts in conjunctive normal form).

There are three functions defined in this file: cnf and dnf reduce propositional formulas to conjunctive and disjunctive normal forms correspondingly, is id(x) is 1 if x is identity (tautology) or 0, if it is not identity.

The strategy can ord is used for the reduction of logic formulas. Two systems of rewriting rules are used in both cases. Really the first system could be taken equal to the second. But the use of additional rules makes the reduction faster.

6 Polynomials

The following example of ap-module named pl ac.ap refers to computer algebra. It represents the algorithm of expanding polynomials represented in natural form that is constructed by means of arithmetical operations sum and multiplication (other representations are recursive representations, representations using vectors of exponents for monomials, vectors for coefficients and so on) (Example 8).

```
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)
);
```

Really the function canpl is realised as an interpreter of unary operation, but before it was prototyped as a simple algebraic program. To use rationals for coefficients an additional mechanism for polymorphic functions is used.

To understand this mechanism we must return to more detailed description of interpreters of operations. There are three levels of interpreters:

- Internal interpreters implemented as C programs;
- Interpreters for polymorphic operations defined through the table polist visible and controlled by a user;
- Extended interpreters introduced by a user by means of a rewriting rule system extcan.

These interpreters are called by function can. The argument of this function is a node of a module state with the decomposition t • ˜(t_1, . . . , t_n). The operation ˜ defines the internal interpreter •˜ written in C. The application of this interpreter to a term t can be successful or not. If the application is successful a tem t is changed to •˜(t), otherwise the polymorphic interpreter is called. At this moment the table polist is used. In the example considered here the cite of a table is the following file rat.ap.

```
INCLUDE <std.ap>
INCLUDE <gen_obj.ap>

NAME rational,
    polist,simp,israt,
    deration, prn_rat;
```

```
MARKS rat(2),

   quote(2,57,"//"),   /* integral part of a quotient */
   GCD(2); /* greatest common devisor */

rational:=1;

polist:=ARRAY(
  /* Polymorphic functions */
  (()+(),add),
  (()-(),sub),
  (()*(),mul),
  (()/(),div),
  (()^(),pwc),
  (()<(),lt),
  (()>(),gt),
  (()<=(),lte),
  (()>=(),gte),

  /* Polymorphic data constructors */
  (rat((),()),0)
);

extcan:=nil;

simp:=rs(x,y,z)(
  rat(x,y) = simp(rat(x,y),GCD(x,y)),
  (z==y) -> ((rat(x,y),z) = x//z),
  (rat(x,y),z) = rat(x//z,y//z)
);

deration:=rs(x,y)(
  x$y = y*x,
  rat(x,1) = x,
  rat(x,y) = x / y
);

prn_rat:=proc(T)loc(S)(
  T-->copy(T);
  rational->(
    S:=T;
    markcan(()/(),can0);
    nbt(S,deration);
```

```
                                    prn(S);
                                    markcan(()/(),div_rat)
                                  ) else prn(T)
                                );

                                israt:=rs(x,y)(
                                  rat(x,y) = 1,
                                  x = 0
                                );

/* ######################## Specifications ################################
                                cor_sg:=rs(x,y)(
                                  (y<0) -> (rat(x,y) = rat((-1)*x,(-1)*y))
                                );

                                GCD:=rs(x,y)(
                                  (x,0) = x,
                                  (0,x) = x,
                                  (x > y)->((x,y) = GCD(y,x mod y)),
                                  (x <= y)->((x,y) = GCD(x,y mod x))
                                );

                                Abs:=rs(x)(
                                  (x<0)->(x = (-1)*x)
                                );

                                add:=rs(x1,x2,y1,y2)(
                                  rat(x1,y1)+rat(x2,y2) = simp(rat(x1*y2+x2*y1,y1*y2)),
                                  isint(x1)->(x1+rat(x2,y2) = simp(rat(x1*y2+x2,y2))),
                                  isint(x2)->(rat(x1,y1)+x2 = simp(rat(x1+x2*y1,y1)))
                                );

                                subt:=rs(x1,x2,y1,y2)(
                                  rat(x1,y1)-rat(x2,y2) = simp(rat(x1*y2-x2*y1,y1*y2)),
                                  isint(x1)->(x1-rat(x2,y2) = simp(rat(x1*y2-x2,y2))),
                                  isint(x2)->(rat(x1,y1)-x2 = simp(rat(x1-x2*y1,y1)))
                                );

                                mul:=rs(x1,x2,y1,y2)(
                                  rat(x1,y1)*rat(x2,y2) = simp(rat(x1*x2,y1*y2)),
                                  isint(x1)->(x1*rat(x2,y2) = simp(rat(x1*x2,y2))),
                                  isint(x2)->(rat(x1,y1)*x2 = simp(rat(x1*x2,y1)))
                                );
```

```
div:=rs(x1,x2,y1,y2)(
  rat(x1,y1)/rat(x2,y2) = cor_sg(simp(rat(x1*y2,x2*y1))),
  isint(x1)->(x1/rat(x2,y2) = cor_sg(simp(rat(x1*y2,x2)))),
  isint(x2)->(rat(x1,y1)/x2 = cor_sg(simp(rat(x1,y1*x2)))),
  (isint(x1)&isint(x2)) ->
     (x1/x2 = cor_sg(simp(rat(x1,x2))))
);

pwc:=rs(x1,x2,y1)(
  (isint(x2)&(x2>0))->
  (rat(x1,y1)^x2 = simp(rat(x1^x2,y1^x2))),
  (isint(x2)&(x2<0))->
     (rat(x1,y1)^x2 = simp(rat(y1^((-1)*x2),x1^((-1)*x2))))
);

lt:=rs(x,y)(
  (x < y) = tst(subt(y-x))
);

gt:=rs(x,y)(
  (x > y) = tst(subt(x-y))
);

tst:=rs(x,y)(
  rat(x,y) = (x>0),
  x = (x>0)
);
```

*/

The table polist contains two types of lines: polymorphic operations with the names of corresponding interpreters and polymorphic data constructors. The call of polymorphic interpreter is performed only if one of the arguments of t is built by means of a polymorphic data structure. The names of polymorphic interpreter can be an atom naming an internal procedure or the name of a function written by a user in APLAN. If polymorphic interpreter gets the negative result, then the system extcan is applied to t as a third level interpreter. The system extcan can be used for the user definition of external interpreter independently of polymorphism. For this purpose a statement

markcan(< typeofoperation >, ec)

After this statement extcan will be called as a first level interpreter for a given operation.

7 Logic programming

In this section a universal problem solver developed in the spirit of constraint logic programming will be considered. The solver described below searches for the solution of a problem on a subject domain defined by a set of axioms, which are quantifierless formulas with variables assumed to be tied by universal quantifiers. Each formula is supposed to be elementary (that is atomary formula or the negation of atomary formula) or represented as implication $P \Longrightarrow Q$ 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), . . .}.

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), . . .} and X is a V –context with values in T(A, V) that is a substitution of a type {V (1) • t_1, . . . , V (n) • t_n}, t_1, . . . , $t_n \in$ T(A, V). A solution of an elementary problem (P,X) is a new V -context Y = {V (1) • s_1, . . . , V (n) • s_m} such, that m • n, si is an instance of ti for i = 1, . . . , 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 | 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 | 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 AB 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 constants 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 of the solver, may be described in the terms of three-valued logic of Klinee on the base of the paper [4] as it was done in [3].

References

[1] S.V.Konozenko A.A.Letichevsky, J.V.Kapitonova. Computations in aps. Theoretical Computer Science, 119:145–171, 1993.

[2] J.V.Kapitonova, A.A.Letichevsky, M.S.L'vov, and V.A.Volkov. Tools for solving problems in the scope of algebraic programming. LNCS V.958, pages 31–46. Springer-Verlag, 1995.

[3] A.A.Letichevsky J.V.Kapitonova. On constructive mathematical descriptions of subject domains. Kibernetica, (4):17–35, 1988.

[4] M.Fitting. A kripke-kleene semantics for logic programs. J.Logic Programming, (4):295–312, 1985.