UDC 681.142.2

APS C++ User's Library

A. Letichevsky, A. Letichevsky Jr., V. Peschanenko

LitSoft, 14 Rileeva Str., ap. 2, Kyiv, Ukraine,
e-mail:vladimirius@gmail.com

The general information about the APS algebraic programming system (terms rewriting system) is briefly described in the present article. It is justified practical necessity of creation of APS C++ User's Library, its conception is presented and its main functions are listed. Also it is mentioned few words about the translator APLAN-C++.

Коротко описаны общие сведения о системе алгебраического программирования APS(системе переписывания термов). Обосновано практическая необходимость в создании библиотеки APS C++ User's Library, приведена ее концепция и перечислены основные функции. Упомянуто о трансляторе APLAN-C++.

Introduction

Algebraic programming – is programming, based upon rewriting. Algebraic programming is an amplification of functional programming and it is applied at computer algebra tasks solution (such as the problem of words in finite definite algebras, augmenter algorithms of Knout-Benedix or Buchberger) as well as the tasks related to operational semantics of programming languages (performed algebraic specifications of programming software components, definition of operational semantics of programming languages, development of interpreters and prototypes of programming software components etc).

The APS algebraic programming system was developed in the Institute of Cybernetics in V.M. Glushkov honor of the National Academy of Sciences of Ukraine [1], as an instrumental tool for creation of applied system of algebraic programming.

In contrast to traditional approach, oriented to usage of canonical rewriting rules systems with the "obvious" strategy of their application, in APS it is possible the combination of any rewriting rules systems as well as different strategies of rewriting.

Such approach amplifies considerably the possibilities of rewriting techniques as their flexibility and expressiveness rise. APS integrates four basic paradigms of programming in such a way that the main part of the program can be written in a view of rewriting systems, the imperative and functional programming is used for determination of strategies, the logic paradigm is realized on the base of rewriting, using a build-in procedure of unification [2].

Originally the algebraic programming system was created by the authors as a system of algorithms prototyping. However the experience of usage of this system in some commercial projects has shown that it can be and should be used as a commercial system not only for the prototyping of algorithms, but also for their final realization and implementation in big commercial products.

Traditions of algebraic programming and usage of APS system as in commercial as well as in research projects were always supported by the company LitSoft [3], the Institute of Cybernetics in V.M.Glushkov honour of the National Academy of Sciences of Ukraine [1] as well as by the Kherson State University [4].

Programming in APS algebraic programming system is performed on two levels: the upper level – language of APLAN algebraic programming system, the lower level – language of realization of the system C++ itself. The process of development of programs in system of algebraic programming passes several steps: step of building of prototype at APLAN language, step of analysis of this prototype and the step of realization of the final version in language of the lower level of C++.

Certainly, APS has a number of imperfections. One of the main ones is the fact the language APLAN is a scripting language and its procedural part is slow, as a result of this for realization of final versions mainly the most critical parts of the algorithms are transferred to C++. In this connection the questions of automation of this process will always be actual.

The transfer of prototype in APLAN language to realization of final version is rather complicated, in spite of the fact that there were several attempts to speed up this process. [5]. However they contain a number of imperfects, so now in practice they are not used. As a result of it now this process is not automated. From this appears important task: how to automate the process of transfer from the prototype in APLAN language to realization of final version and what is needed for it?

In a rather spread analog of APS – in MAUDE system such process is already automated, but only for very limited part of the language of MAUDE system, because in our opinion the automation of the process of development will lead the APS programming system to a new level, that will give possibility to talk about APS as one the most powerful tools in development of complicated algorithmic systems [6].

For automation of this process with the usage of APS we have decided to mark two steps: step of realization of the User's Library APLANC and the step of realization of convector from the language APLAN into the language C++.

APS

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.

$$\begin{aligned} &F(0) = 1,\\ &F(1) = 1,\\ &F(n) = F(n-1) + F(n-2) \end{aligned}.$$

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 < *std.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 *std.ap*. Now the name of a system must be defined

NAME R;

and assigned a value by means of initial assignment:

$$\begin{aligned} &R := rs(n)(\\ &\quad F(0) = 1,\\ &\quad F(1) = 1,\\ &\quad F(n) = F(n-1) + F(n-2)\\ &);\end{aligned}$$

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

$$\begin{aligned} T &= ((F(7)+F(6))+F(7))+F(8) =\\ &= (((F(6)+F(5))+F(6))+F(7))+F(8) =\\ &\quad \ldots\ldots\ldots\ldots\ldots\ldots\\ &= (...((F(1)+F(0))+F(1))+....)+F(8). \end{aligned}$$

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:

$$\begin{aligned} T &= (...((1+F(0))+F(1))+....)+F(8) =\\ &= (...((1+1)+F(1))+...)+F(8). \end{aligned}$$

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 ntb (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:=ntb(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 it is sufficient to input a text file *fib.ap* which contains this task and execute it using command line

```
aps.exe –i  fib.ap
```

This command interprets a procedure task in this module.

Let us consider now how the strategies ntb and nbt 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 $f(x_1, \ldots, x_n)$, where f is an operation of arity n, then this node is connected by edges numbered by $1, \ldots, n$ with the nodes corresponding to the subexpressions $x_1, \ldots, x_n$.

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 atom F, the second argument is an expression x.

Both strategies ntb and nbt 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 ntb (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 nbt (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 nbt will compute the value of an expression while the strategy ntb 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. 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.

```
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 $b = 1$:

$$a*(x+y)+b*x=(a+b)*x+y,$$
$$a*(x+y)+x=(a+1)*x+y,$$
$$(x+y)+x=2*x+y$$

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 ntb and nbt 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

```
R1 := rs(n)(
  a*(x+y)+b*x=(a+b)*x+a*y,
  a*(x+y)+x=(a+1)*x+a*y,
  (x+y)+x=2*x+y,
  F(0) = 1,
  F(1) = 1,
  (isint(n) & (n > 0))-> (F(n) = F(n-1) + F(n-2))
);
```

and new task is:

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

To write this system in APLAN the following statement must be written first

INCLUDE < *strat.ap* > .

This statement includes some standard definitions of additional strategies and especially provides the use of compile rewriting system, which will be use for additional APLAN language statements.

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

$$a \ast \mathrm{F}(n) + b \ast \mathrm{F}(n-1).$$

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

$$\text{if } n > 0 \text{ then}$$
$$F(n) = 1 * F(n) + 0 * F(n-1) = f(1, 0, n),$$
$$f(a, b, n) = f(a + b, a, n - 1).$$

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

$$\begin{aligned} &R2 := rs(n)(\\ &\quad f(a,b,0) = b, \\ &\quad f(a,b,n) = f(a+b,a,n-1), \\ &\quad F(n) = f(1,0,n) \\ &); \end{aligned}$$

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 [2].

APLANC

For automaton of the process of development we introduce the notion APLANC of the language. It is a number of procedures, written in language C++, which simplify the work with the internal structures of APS system. This set of functions exactly will be used for transfer of program from the upper level of APLAN language to the lower level of realization of system C++.

The structure of the final version of the program. It is important to see how the program transferred from APLAN to C++ will look like. So, the requirements for the future convector of APLAN-C++ are the following:

Generated C++ code should depend on the clue library;

Generated C++ code should use the functions of APLANC only;

All necessary canonizators of marks (except the basic ones, determined by the documentation) are sunk only by hand only at the discretion of the Library User;

The convector generates the code in such a way that the user is lack only to write the corresponding makefile for the project and to compel it.

So, the APLANC library includes the following functions:

Marks and the basic canonizators;

Functions of construction of tree;

Some operators of the upper level;

Functions of work with rewriting machine;

Conception APLAN-C++ of the translator.

We should draw your attention to the fact that the access function to the concrete marks is not presented to the user, that will considerably simplify the understanding of C++ program in our opinion. But all necessary for the checking of the marks will be realized in the 3rd item.

The marks and the main canonizators. The total table of marks of the APS algebraic programming system is used for initialization of the new program clue.

Canonizators of the marks of APS system – is one of the powerful means at designing of programs in APLAN language. Certainly, the library of APLANC functions is a minimal set of standard system canonizators: canonizaors of all arithmetic systems(+,-,*,/,^), logic operations (&,|/,~), function mrg and canonizator. The more detailed information can be received herein [ ].

Functions of trees construction. The functions of trees construction include: make_formula, make_hash_formula.

bool make_fomula(const char* cc,int n,…); – the function uses internal parser of APLAN language with a purpose to build a tree by the line ss, if the tree hasn't been constructed, false returns, if not it is performed substitution n-times instead of each entrance () from the left to the right of the corresponding arguments.

bool make_hash_fomula(unsigned long hash_hum,const char* cc,int n,…) – the function checks if the term with such hush number hash_hum exists, if it exists, the indicative tops of trees return. If not, the function constructs a tree and adds its top to the hash table. FALSE returns only when the tree is not constructed yet and it is not in a hash table.

Example:

```
node_ptr x = make_formula("a-b",0);          (1)
node_ptr y = make_formula("c-d",0);          (2)
node_ptr z = make_formula("()+2+()",2,*x,*y); (3)
```

1) – we build a tree for expression a-b, 2) – we build a tree for expression, 3) – we build a tree for expression ()+2+() (the empty brackets are considered the mark Nil), then from left to right we make a substitution instead of () firstly term x, then term y. As a result in z it will be a tree for expression (a-b)+2+(c-d).

These two functions are the main functions at transfer of the program to the lower level.

Some operators of the lower level. In some cases it is rather convenient to have some analogs of APLAN language operators. It is quite difficult to present the work with canonizators without mark_can – the function of installation of canonizator for corresponding mark. Function-canonizator should look like int can_name(clew_ptr &_clew,node_ptr &arg). The function returns the error number, and 0 – if the function has been working normal. From this very important conclusion can be made – so far as in the capacity of canonizator the rewriting rules system can be used, its headline should look like: int can_name(clew_ptr &_clew,node_ptr &arg).

For the realization of rewriting machine in APLAN we need the function let and here is its syntaxes:

bool let(node_ptr &nd,const char *cc,nodes_ptr &res);

The function returns 0 if the process of comparison wasn't succeeded [], 1 – if not. It is very important for the library of functions APLANC if the process of comparison succeeded, to receive immediately the subtrees, which are necessary for the further work. So, especially it was introduced the name ac_h (APLANC here), such subtrees, that reply in the line to this name and are copied to the structure nodes_ptr []. The size corresponds to the quality of entrances of fc_h in ss, the order is from left to right.

Example:

```
node_ptr x = make_formula("x+y+2+3",0);   (1)
nodes_ptr args;                           (2)
if (let(x,"_+_+ac_h+ac_h",args)){         (3)
  prn(args.size());                       (4)
  prn(args[0]);                           (5)
        prn(args[1]);                     (6)
}
```

1) – see above, 2) – declaration of tops list, 3) – if the process of comparison succeeded, i.e the term has three marks +, the tree is right-side, then in args it will be written two last summands, i.e. 2 and 3., 4), 5), 6) – screen output.

Working functions with rewriting machine. It is known that one of the imperfects of APS system if low interpretation of the procedural part of APLAN language, so in our opinion the hole rewriting machine APS and its working functions (strategies) move to APLANC [].

However the translator APLANC++ itself can work without rewriting machine if the rewriting rules system will be translated into function of C++ language (the headline of such function has been described earlier). The realization of the necessary strategies also doesn't cause difficulties in this case. The process of transfer itself from the rewriting rules system to C++ functions is going to be described in our further publications.

The rewriting machine of APS algebraic programming system transfers the rewriting rules system preliminary into special commands language, and then it interprets.

APLAN-C++ translator's conception. The APLAN-C++ translator's conception is a trade secret for the moment and it can't be discussed in the research paper. We would like to note that this translator uses the APLAN library and till the moment of issue of the present paper we plan to realize it at 70%.

Conclusion

The APLANC User's library is a powerful tool of automation of transfer process in APLAN to the final version in C++. Its realization will considerably amplify the spheres of application, it will allow to debug C++ programs quickly and professionally as per the prototype as well as per the final version, that will shorten the terms of development of complicated algorithmic systems.

I express my thanks to the company LitSoft for the support of symbolic conversions in Ukraine represented by Mr. A.A. Letichevskiy (jr.), our ideological inspirer Mr. A.A. Letichevskiy as well as to my wife Mrs. Marina Y. Peschanenko for the professional reading of the present article.

1. *Glushkov* Institute of Cybernetics NAS Ukraine [http://www.icyb.kiev.ua].
2. *Letichevsky A.A., Kapitonova J.V., Volkov V.A., Chugajenko A., Chomenko V.* Algebraic Programming System APS (user's manual). [http://aps.ksu.ks.ua/files/6.zip].
3. *LitSoft* [http://alimp.kiev.ua].
4. *Kherson* State University [http://www.ksu.ks.ua].
5. *Песчаненко В.С.* Об одном подходе к проектированию алгебраических типов данных // Проблеми програмування. – 2006. – №2-3. – С. 626-634.
6. *The* MAUDE System [http://maude.cs.uiuc.edu].