Theoretical Computer Science 119 (1993) 145–171
Elsevier

Computations in APS

A.A. Letichevsky, J.V. Kapitonova and S.V. Konozenko
Glushkov Institute of Cybernetics, Ukrainian Academy of Sciences, Kiev 252207, Ukraine

Abstract

Letichevsky, A.A., J.V. Kapitonova and S.V. Konozenko, Computations in APS, Theoretical Computer Science 119 (1993) 145–171.

An algebraic programming system (APS) integrates four main paradigms of computations: procedural, functional, algebraic (rewriting rules) and logical. All of them may be used in different combinations at different levels of implementation. Formal models used in the developing computational techniques for APS are presented and discussed. These include data structures, algebraic modules, rewriting and computing, canonical forms, tools for building strategies and data types.

1. Introduction

An algebraic programming system (APS) is under development in the Glushkov Institute of Cybernetics of the Ukrainian Academy of Sciences. It is a professionally oriented instrumental tool for the design of applied systems based on algebraic and logical models of subject domains. The main programming technique used in the system is rewriting-rule-based programming.

Rewriting technique has been intensively studied [7, 5]. There are many implementations of term rewriting systems. Some of them support algebraic specifications (ASF [1], ASSPEGIQUE [2]), others are rewriting laboratories based on the Knuth–Bendix algorithm for computing canonical systems from a set of equational axioms (REVEUR3 [6], for instance). The languages of the OBJ family [3] and O'Donnell's languages [11] are the bases for equational programming. Rewriting technique was used in ANALITIC [13], which is in some sense the prototype of APS. It is used in Mathematica^{TM} [14] and other computer algebra systems.

Unlike a traditional approach, which is oriented to the use of canonical systems of rewriting rules with "transparent" strategy for their application, in APS it is possible to combine arbitrary systems of rewriting rules with different strategies of rewriting.

Correspondence to: A.A. Letichevsky, Glushkov Institute of Cybernetics, Ukrainian Academy of Sciences, Kiev 252207, Ukraine. Email: let%d105.icyb.kiev.ua@relay.USSR.EU.net.

Such an approach essentially extends the possibilities of rewriting technique and enlarges its flexibility and expressibility. The main features of the experimental version of the system APS-1 implemented on IBM PC were described in [10, 8]. The methodology of research may be expressed by the following principles.

(1) Integration of four main paradigms of programming: procedural, functional, algebraic and logical. This integration is achieved by the adjusted use of corresponding computational mechanisms.

(2) The use of object-oriented methods and ideas of large grain parallelism to organize complex programs and system implementation. As an example the implementation of APS on multiprocessor systems is now studied.

(3) Support of the evolutionary development of the system. This includes the modularity and hierarchical structure of the system, localization of different computational mechanisms and the possibility of step-by-step transfer of complexity from high to low levels to achieve more efficiency. The final goal of evolution is the creation of effective problem solvers on mathematical models of subject domains.

This paper describes the main computational mechanisms of APS as well as their interaction and use. These mechanisms include strategies of rewriting, canonical forms, data types, recursive data structures processing, inheritance and interaction between modules. The description is based on mathematical models which were built to ground decisions and to develop the main algorithms in the system.

2. Example of an algebraic program

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

```
INCLUDE ⟨ac.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
);
NAME cnf;
cnf := proc(x)(
       ntb(x,R),
       can_ord(x,R1,Q1),
       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;
Id := rs(x,y,z)(
      1 → 0=0,
      0 → x=1,
      x → x=1,
      x → 1=1,
      x → y || z=x & deM(~(y))→z,
      x → y & z=(x→y) & (x→z),
      x || y → z=(x→z) & (y→z),
      x & y → ~(z)=subs(z=1,x)→deM(~(subs(z=1,y))),
      x & y → z=subs(z=0,x)→deM(~(subs(z=0,y))),
      x → y=0
);
```

```
NAME ntb2;
ntb2 := proc(t, R)(
        appls(t, R);
        (ART(t) > 0) → ntb2(arg(t, 1), R);
        t := can(t);
        ntb2(t, R)
);
NAME is_id;
is_id := proc(x)(
        ntb(x, R);
        x - →(1 → x);
        ntb2(x, Id);
        return(x)
);
```

The first sentence of log.ap means that it includes the previously defined algebraic program ac.ap which contains some standard definitions and a description of associative–commutative operations. Especially, it contains the description of logical connections ~ (negation), &, || (disjunction), →, ⇔, and information that defines & and || as boolean operations. The next sentence introduces a new operation symbol subs with arity 2. Different interpreters may be used to evaluate algebraic programs. The interpreter nsint which is to be used for evaluating log.ap interprets subs as a substitution function: subs$((x_1 = y_1, \ldots, x_n = y_n), z)$ substitutes $y_1, \ldots, y_n$ instead of all occurrences $x_1, \ldots, x_n$ to z where $x_1, \ldots, x_n$ are supposed to be different atoms.

The values of names R, Rl, Ql, deM and Id defined by initial assignments are systems of rewriting rules. To apply them to algebraic expressions (terms) one may use standard strategies implemented by a current interpreter or write one's own. Function cnf computes (some) conjunctive normal form of logical expression. It is defined by means of a procedure that uses two standard strategies ntb and can_ord. The first is a one-time top-bottom strategy. It passes over the nodes of expressions representing trees in a top-bottom manner and checks the applicability of rewriting rules in the order they are written in the system. This strategy is used for the elimination of → and ⇔ and transferring negations by de Morgan rules.

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

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

3. The structure of APS

Let us consider the main notions of APS.

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

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. The syntax and (informal) semantics of this language were described in [9] and will be discussed later. Each program contains the description of signature Ω with a syntax for constructing algebraic expressions (terms). It also defines the set of names X and atoms A. These objects, together with numbers and strings, constitute the set Z of *primary objects.* The three sets mentioned above define the type (Ω, X, A) of ap-module. The types of ap-modules are partially ordered by the inclusion relation (symbols of Ω are considered jointly with their descriptions, which include, in particular, the arity of each symbol). If $(\Omega, X, A) \subset (\Omega', X', A')$, ap-module M of the type (Ω', X', A') is said to belong to the class $C(\Omega, X, A)$. Two classes are said to be *compatible* if they have a common lower bound that is a common subclass. Parameters of this subclass contain parameters of both compatible classes. The algebraic program also defines the initial values of the names. These values are objects of the type $T_\Omega(Z)$.

Algebraic modules contain internal representations of the data structures defined in ap-modules. They are being created by system commands that refer to ap-modules as new object generators. Algebraic module M generated by program P inherits its type and the initial values of its names. The notion of an a-module is a dynamical one. It has a *state* which may change in time. The change of state of an 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 $CA(\Omega, X, A)$ for them are defined similarly to the corresponding notions for ap-modules. Thus, the ap-modules play the same role w.r.t. a-modules as the classes w.r.t. the objects in the object-oriented programming.

States. The state of an a-module of the type (Ω, X, A) consists of two components. First is the memory state, i.e. the mapping $\sigma: X \to T = T_\Omega^*(Z)$. The second component expresses the possibility for data to have common parts. Instead of the notion of a reference or pointer the more abstract notion of *node equivalence* will be used. To define this equivalence we use the notions of occurrence of term and subterm defined by it, which are well known in the theory of rewriting.

The occurrence is a sequence $(i_1, \ldots, i_n)$ (which may be empty) of positive integers. The set of occurrences $O(t)$ for term t is defined jointly with the function $arg: S \to T$ where $S \subset T \times N^*$ such that $(t, p) \in S \Leftrightarrow p \in O(t)$ by the following recursive definition:

(1) $(\ ) \in O(t)$ and $arg(t, (\ )) = t$.

(2) $p \in O(t)$, $arg(t, p) = \omega(t_1, \ldots, t_n) \Rightarrow (p, i) \in O(t)$, $arg(t, (p, i)) = t_i$, $i = 1, \ldots, n$.

These definitions work for finite as well as for infinite terms. The sign "," in occurrences is used as a binary associative operation. Any term is uniquely defined by the set $O(t)$ and function $root(t, p)$ defined by equalities:

$$arg(t, p) = \omega(t_1, \ldots, t_n) \Rightarrow root(t, p) = \omega;$$

$$arg(t, p) \in Z \Rightarrow root(t, p) = arg(t, p).$$

Let $\sigma: X \to T$ be the memory state. Define X-occurrence as a pair (x, p) where $x \in X$, p is occurrence and the set $O(\sigma)$ of X-occurrences for σ, so that $(x, p) \in O(\sigma) \Leftrightarrow p \in O(\sigma(x))$. Now define the *node equivalence* ε for the state σ as an equivalence relation on the set $O(\sigma)$ satisfying the following axioms:

(1) $(x, p) = (x', p')(\varepsilon) \Rightarrow arg(\sigma, p) = arg(\sigma', p')$;

(2) $(x, p) = (x', p')(\varepsilon)$, $arg(\sigma(x), p) = \omega(t_1, \ldots, t_n) \Rightarrow (x, (p, i)) = (x', (p', i))(\varepsilon)$, $i = 1, \ldots, n$;

(3) node equivalence is a finite index, that is, it has only a finite number of equivalence classes.

Equivalence $(x, p) = (x', p')(\varepsilon)$ means that subterms $arg(\sigma, p)$ and $arg(\sigma', p')$ have a common root node. Another representation of the module state will be considered below.

System interpreters. These are programs destined for interpretation of the procedures written in APLAN. They are developed in C language on the base of libraries of functions and data structures to work with internal representation of system data structures. The corresponding extension of the C language is called L2C. Each interpreter is connected with the distinct type (Ω, X, A) which defines the classes $CI(\Omega, X, A)$ to which the interpreter belongs in a similar manner as for modules. This type defines the restricted algebraic modules which can be executed by the given interpreter. All of them must belong to the class which is compatible with the class $CA(\Omega, X, A)$.

Each interpreter specifies the operational semantics of APLAN for the given class of a-modules 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 a more detailed classification w.r.t. to their semantical properties.

Components of the system. The main component is the naming of system objects, i.e., the set of ap-modules, a-modules and interpreter names together with their values. The shell of the system provides the interface of the user with the following subsystems:

- control system for problem solving by means of system commands and existing algebraic programs;
- algebraic programs development system;
- interpreter development system.

System commands provide the possibility of making up the following actions:

- creating a new a-module x by means of the program y: "create xy", 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 the program y: "complete xy", x and y are the names of existing files;
- executing the procedure x of the algebraic module y by means of interpreter z: "zxy", z is the name of an executable file, x is the name defined in a-module y.

System commands may be executed when required by the user or used as internal calls in algebraic programs. Such calls, along with some additional possibilities, provide interactions among algebraic modules.

4. APLAN

Syntax. An algebraic program is defined as a sequence of sentences. The following types of sentences exist:

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

```
⟨inclusion⟩ ::= INCLUDE ⟨file name inserted into "⟨ ⟩"⟩
  | INCLUDE "⟨file name⟩"
```

Comments are indicated by /* */. Strings are symbol sequences inserted into " ".

Semantics. An algebraic program has two different meanings. The first meaning corresponds to an ap-module considered as a generator of new algebraic modules and is defined by *generic semantics*. The second meaning depends on the interpreter being used. It is defined by *operational semantics* and may vary within wide limits.

Generic semantics. Realized by system commands `create` and `complete`. The a-module is created or completed by means of sequential processing of the sentences that constitute the ap-module. Inclusion sentence `INCLUDE` x means that the 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 Ω of the a-module. Besides the algebraic operations themselves, marks may be used as a function or to 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 a mark description it may be used for infix representation of expressions. In this case prioritizing helps us to omit some brackets. When the arity is UNDEF, the mark may be used with a different arity >0 (this mark may be associated with an 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. This application always has the highest priority in the system. Atoms are identifiers which occur in a program and which were not described as names or marks or infix notations.

Internal representations of algebraic expressions are Ω-trees constructed in an obvious way. After processing of name descriptions the value of each name is initialized by an empty object which is the 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 in the case when the name z follows the symbol VAL. In this case the value of z will be referenced instead of VAL z. The use of this tool makes it possible to identify the nodes of internal representations of trees. If VAL $z = arg(y, p)$, then after this assignment the equivalence $arg(x, p) = arg(z, ())(\varepsilon)$ will appear.

When the name or mark is redefined the previous definition is cancelled. The same is true for the initial assignment. Thus it means that it is impossible to create objects with loops. Indeed, after the initial assignment $x := \ldots \text{VAL}\, x \ldots$, all occurrences of VAL x will be replaced by an empty object even if x was already initialized.

5. Operational semantics

The operational semantics of APLAN is implemented by interpreters. Each interpreter contains three main computational mechanisms:

- procedure interpreter,
- interpreter of operations,
- interpreter of internal calls.

All interpreters in the system are extensions of the minimal interpreter sint which has the type (Ω_0, X_0, A_0) (standard interpreter) and is described below. The signature and names of sint are the standard ones. Descriptions of standard operations and names are contained in the ap-module std.ap, which particularly includes the following descriptions:

```
MARKS
  /* Arithmetical and algebraic operations and functions */
  POW(2,60,"^"), M(2,58,"*"), DIV(2,57,"/"),
  ADD(2,55,"+"), SUB(2,54,"-"),
  /* Predicates */
  LE(2,40,"<="), LS(2,40,"<"), ME(2,40,">="),
  MR(2,40,">"), EQ(2,11,"=="), EQU(2,11,"="),
  /* Logical connections */
  ~(1,30), AND(2,29,"&"), OR(2,28,"||"),
  IFF(2,26,"<=>"), IF(2,18,"->"),
  /* Separators */ L(2,7,","), LL(2,5,";"),
```

```
/* L2B operations */
SET(2,20,"-→"), ASS(2,20,":="), ELSE(2,19,"else"),
do(1), while(2),
/* Special functions */
arg(2,61,"arg"), '(1), ART(1), CAN(1), vl(1);
```

Procedures. The procedures of APLAN are algebraic expressions which meet the following syntax:

```
⟨procedure⟩ ::= ⟨sequence of statements separated by "," or ";"⟩
⟨parametrized procedure⟩ ::= proc(⟨formal parameters list⟩)
  ⟨local names⟩ ⟨statement⟩
⟨local names⟩ ::= loc(⟨local names list⟩) | ⟨empty⟩
⟨formal parameter⟩ ::= ⟨identifier⟩
⟨statement⟩ ::= ⟨basic statement⟩ | ⟨conditional statement⟩
  | ⟨while statement⟩ | ⟨do statement⟩ | ⟨internal call⟩
  | ⟨external call⟩ | return | return(⟨algebraic expression⟩)
  | (⟨procedure⟩)
⟨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 name⟩(⟨actual parameter list⟩)
⟨internal name⟩ ::= ⟨atom⟩
⟨external call⟩ ::= ⟨name⟩(⟨actual parameter list⟩)
⟨actual parameter⟩ ::= ⟨algebraic expression⟩
```

Procedures may be the values of names and are evaluated by a procedure interpreter which is the same for all system interpreters. It calls operation interpreters and interpreters of internal calls for evaluating the values of expressions and internal calls, respectively. Semantics of conditional and while statements are usual. Statement do(x) executes the value of name x which must be the sequence of statements. The value of name x in external call $x(y_1, \ldots, y_n)$ must be a parameterized procedure proc(...)... which is evaluated after transferring the actual parameters.

To explain more precisely the evaluation of expressions, semantics of basic statements and transferring parameters, a formal model for the representation of module states must be introduced.

Representation of states. Let us consider the state (σ, ε) for the module of the type (Ω, X, A). Let $U = \{u_1, \ldots, u_m\}$ be the alphabet of symbols set to one-to-one

correspondence with the classes of ε. 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 will also be considered as nodes of a graph representing the set of data structures contained in the given module in the current state.

Let $(x, p) \in u$, $\sigma(x) = t$, $arg(t, p) = \omega(t_1, \ldots, t_n)$. Then if $(x, (p, i)) \in v_i$, $i = 1, \ldots, n$ we shall write $u \to \omega(v_1, \ldots, v_n)$ and call this expression the *decomposition of the node u.* If $arg(t, p) \in Z$ the decomposition of u is $u \to arg(t, p)$. 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 $x \to u$ will be called the *decomposition of the name x.* The set of decompositions for all the nodes and names is called the *node representation of the state* (σ, ε).

It may be shown that the *node representation uniquely defines the state it represents.* Indeed, let $\{u_1 \to s_1, \ldots, u_m \to s_m, x_1 \to v_1, \ldots, x_n \to v_n\}$ be the representation of state (σ, ε). Define substitution $\tau = [u_1 \leftarrow s_1, \ldots, u_m \leftarrow s_m]$. Then $\sigma(x_i) = v_i \tau^k$ for sufficiently large k if $\sigma(x_i)$ is finite or is the limit of $v_i \tau^k$ if this value is infinite. The conditions used to construct decompositions of nodes make is possible to inductively define the node u such that $(x, p) \in u$ for arbitrary X-occurrence (x, p). Conversely, the *node representation is unique up to the renaming of classes.*

Now let $\{u_1 \to s_1, \ldots, u_m \to s_m, x_1 \to v_1, \ldots, x_n \to v_n\}$ be an arbitrary set of decompositions of the type (Ω, X, A) over the node alphabet U such that any node and name has one and only one decomposition. Call this set *clew of data structures.* It may be shown now that *any clew is the representation of some module state.* Indeed, memory state and node equivalence are constructed as was shown above; the axiom of representation is true because $arg(\sigma(x), p)$ depends only on node u such that $(x, p) \in u$.

It is convenient to extend the notion of representation by allowing the rhs's of decompositions to be the arbitrary finite terms over $T_\Omega(Z \cup U)$. Using this extension one may eliminate some wasteful nodes. The node u is called *wasteful* if it occurs in the rhs's of decompositions no more than once. The wasteful node u may be eliminated after replacing its unique occurrence by the 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.*

Theorem 5.1. *There exists a one-to-one correspondence between the states of a module and their minimal (maximal) representations considered up to the renaming of nodes.*

The theorem follows from the statements proved above.

The minimal representation $r = \{u_1 \to s_1, \ldots, u_m \to s_m, x_1 \to v_1, \ldots, x_n \to v_n\}$ of some current state of the module will be fixed later in this section.

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

kinds of values is expressed by the formula

$$Val(t)=(val(t))\tau^{\infty},$$

where τ^{∞} is the limit of τ^k. The second kind of value does not depend on equivalence ε and is used in "invariant" reasoning about algebraic programs. The first kind is used to define precisely the operational semantics of procedural tools. Function $val(t)$ substitutes the values of names and reduces the expression to its main canonical form using interpreters of operations (functions) φ_{ω}. A formal definition includes the following rules:

$$isname(x),\ x\to t\in r \ \Rightarrow\ val(x)=t;$$

$$isfun(f) \ \Rightarrow\ val(f(x))=\varphi_{appl}(nd(f), val(x));$$

$$val(`(t))=t;$$

$$val(\omega(t_1,\ldots,t_n))=\varphi_{\omega}(\omega(val(t_1),\ldots,val(t_n)));$$

$$isname(x),\ x\to t\in r \ \Rightarrow\ nd(x)=nd(t);$$

$$nd(x)=x.$$

Each rule may be applied only if the previous one is not applied. Expression $isfun(f)$ is true if $nd(f)$ is a parameterized procedure or rewriting rule system. Semantics of application is considered in Section 6. Correct computation of the value of algebraic expression t must preserve the current state of a module. It means especially that procedures which may be used as functions must be written without side effects.

Basic statements. In both cases (set and assignment) the value $s\in T_{\Omega}(Z\cup U)$ of the rhs is computed. Consider the set statement. If the lhs is the name x its decomposition is replaced by $x\to s$. Let us consider the statement $arg(x,p)-\!\!\to t$. The value of p must be the sequence $(i_1,\ldots,i_n)$ of positive integers. If $(x,i_1,\ldots,i_{n-1})\in u$ and $u\to q\in r$ then s is substituted instead of the i_nth argument of q. Of course, the arity of q must be greater than or equal to i_n.

Assignment $x:=t$ acts differently. Firstly, 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 the assignment is equivalent to the set statement. Otherwise, if $x\to u$, $u\to q\in r$, decomposition $u\to q$ is changed to $u\to s$.

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

Internal calls. Addressed to procedures implemented on the level of L2C language. The number of formal parameters and how to transfer them (compute values or not) are defined according to specifications of internal procedures.

Invariancy. Each procedure defines the transformation of module states, i.e. computes a function $F(\sigma, \varepsilon)=(\sigma', \varepsilon')$. A procedure is called *invariant* if σ' does not depend on ε. Algebraic procedures, i.e., procedures that compute functions over terms (not over clews) must be invariant. More practical is the notion of *conditional invariancy*, i.e. invariancy on the set of states meeting some given conditions. An important example of such condition is that top nodes of the values of names are all disjoint (in maximal representation all rhs's for the name decompositions are different). Such states are called disjoint. If a procedure does not use set statements and uses only invariant calls it is invariant on the set of disjoint states because assignment is invariant on these states. A stricter condition for states is strong disjointedness. The state is called strongly disjoint if the values of names do not have common parts, i.e. $(x, p)=(x', p')(\varepsilon) \Rightarrow x=x'$.

6. Strategies of rewriting

Rewriting systems. The system of rewriting rules (rewriting system) is the algebraic expression with the following syntax:

```
⟨rewriting system⟩ ::= rs(⟨list of parameters separated by ","⟩)
  (⟨list of rules separated by ","⟩)
⟨rule⟩ ::= ⟨simple rule⟩ | ⟨conditional rule⟩
⟨simple rule⟩ ::= ⟨algebraic expression⟩=⟨algebraic expression⟩
⟨conditional rule⟩ ::= ⟨condition⟩ → ⟨simple rule⟩
⟨parameter⟩ ::= ⟨identifier⟩
```

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 a simple rule is usual; match the lhs with t, if successful then substitute the parameters to the lhs and replace t by the rhs. Before replacement the redex rhs is reduced to its 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 $t=val(\mathtt{x})$, $R=val(\mathtt{y})$. Let z be an auxiliary name with the decomposition $z \to t, x_1, \ldots, x_n$ be parameters of system R, $l=r$ be the rule the lhs of which is matched with t and $u_1, \ldots, u_n$ are z-occurrences corresponding to the values of parameters $x_1, \ldots, x_n$ (nodes of some representation of the current state). If the rule is not left linear, i.e. l has more than one occurrence of some parameter, the first occurrence of this parameter is considered. Substitution of the rhs is then equivalent to the assignment $z:=\mathtt{CAN}(r)$,

the function CAN being defined by the following rules:

$$CAN(x_i)=u_i;$$

$$isfun(f) \Rightarrow CAN(f(x))=\varphi_{appl}(nd(f), x);$$

$$CAN(`(s))=s[x_1 \leftarrow u_1, \ldots, x_n \leftarrow u_n];$$

$$x \in Z \Rightarrow CAN(x)=x;$$

$$CAN(\omega(t_1, \ldots, t_n))=\varphi_\omega(CAN(t_1), \ldots, CAN(t_n)).$$

Conditional rules are applied to terms in the following manner. Matching is done first. If successful the condition is reduced to its main canonical form by computing the function CAN. If the result is 1, applying the rule continues as usual. Otherwise, application is cancelled.

The statement `appls`(t, R) calls `applr`(t, R) while `yes`$=1$.

Semantics of application. If the expression $f(t)$ is a subterm of a term and the function `val` or CAN is computed, first the condition $isfun(f)$ is checked: Is the value of f (computed by `nd`) a functional description or not? There are two types of functional descriptions in APS: procedures and rewriting systems. The evaluation of procedures was described above. Rewriting systems are evaluated by means of procedure `applr`. If $nd(f)=R$ is a rewriting system, then $\varphi_{appl}(R, x)$ may be defined as a value of z after evaluating statements:

```
z := x;
applr(z, R);
```

Function symbol f may occur in some rhs's of R. It means that the system will be called recursively. As an example let us consider the following system:

```
pow := rs(x,y,z)
       (x^1=x,
       x^0=1,
       (x*y)^z=pow(x^z)*pow(y^z),
       (x^y)^z=pow(x^(y*z))
);
```

Function `pow` transforms any term of the form `(x*y*...*z)^n` to `x^n*y^n*...*z^n` and simplifies it if possible.

Language extension. As an example of how to use the functional possibilities of APLAN let us consider a simple mechanism built-in to the procedural interpreter which allows us to easily extend the procedural part of APLAN. There is a system named `compile` which has a rewriting system as a value. When the interpreter meets an unknown statement in the procedure it tries to apply the system `compile` to it. If

the statement remains unknown the interpreter omits it and produces a corresponding message. The current state of the compile system (a piece of ap-module extstd.ap) follows.

```
NAMES compile, conc;
MARKS for(4),
  forall(2),
  forallw(3),
  as(2,8,"assn");
compile := rs(x,y,z,u,i)(
             (arg(arg(x,y),z)-→u)=
               compile(arg(x,conc(y,z))-→u),
             (arg(x,y)-→z)=set(x,y,z),
             (x-→y)=setname(x,y),
             for(x,y,z,u)=(x,while(y,(u,z))),
             forall(x=arg(y,i),z)=
               for(i:=1,1⇐'(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
               ),
            ((x,y)assn z)=(x:=arg(z,1),compile(y assn arg(z,2))),
              (x assn y)=(x:=y),
            dowhile(x,y)=(x;while(y,x))
);
conc := rs(x,y,z)(
          ((x,y),z)=(x,conc(y,z))
        );
```

Canonical forms. A typical approach to algebraic computations to consider algebraic expressions up to some congruence consistent with the identities of the algebra that defines the subject domain. Function CAN may help to realize this idea. It defines the equivalence $t=t'(CAN) \Leftrightarrow CAN(t)=CAN(t')$ which must be the congruence: $t_1=t'_1(CAN),\ldots,t_n=t'_n(CAN) \Rightarrow \omega(t_1,\ldots,t_n)=\omega(t'_1,\ldots,t'_n)(CAN)$. This is equivalent to the existence of function *can* such that

$$CAN(\omega(t_1,\ldots,t_n))=can(\omega(CAN(t_1),\ldots,CAN(t_n))).$$

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:

$$can(\omega(t_1, \ldots, t_n)) = \varphi_\omega(\omega(t_1, \ldots, t_n)).$$

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

Term t is called *normalized* w.r.t. can if $can(s) = s$ for any subterm s of term t.

Theorem 6.1. *The following condition is sufficient for the correctness of* CAN: *if* $t_1, \ldots, t_n$ *are normalized w.r.t.* can *then* $can(\omega(t_1, \ldots, t_n))$ *is also normalized.*

It is obvious that if t is normalized then $CAN(t) = t$ (induction and taking into account that $CAN(z) = z$ *if* $z \in Z$). Therefore $CAN(CAN(t)) = CAN(t)$.

The condition of this theorem reduces the check for the correctness of CAN to analysis of the normalization properties of operation interpreters.

A simple example realized in most of the APS interpreters are operation interpreters that evaluate constant computations for arithmetical and boolean operations implementing some simple identities such as $x + 0 = x$ or $x \parallel 1 = 1$. A more complicated example is the interpreter of binary operation arg which is defined as follows:

$$0 \leqslant i \leqslant n \Rightarrow \varphi_{\mathrm{arg}}(\mathrm{arg}(\omega(t_1, \ldots, t_n), (i, j))) = \varphi_{\mathrm{arg}}(\mathrm{arg}(t_i, j));$$

$$0 \leqslant i \leqslant n \Rightarrow \varphi_{\mathrm{arg}}(\mathrm{arg}(\omega(t_1, \ldots, t_n), i)) = t_i;$$

$$\varphi_{\mathrm{arg}}(t) = t;$$

Basic strategies. General questions on constructing strategies and the notion of local strategy were discussed in [10, 8]. Optimization problems were considered in [9]. Let us consider the main strategies implemented in APS as internal procedures. All of them may also be written as external ones.

Strategy ntb is a one-time top-bottom strategy:

```
NAME ntb;
ntb := proc(t,R)loc(s,i)(
         appls(t,R);
         forall(s=arg(t,i),
           ntb(s,R)
         );
         t:=can(t)
);
```

Strategy nbt is a one-time bottom-up strategy.

```
NAME nbt;
nbt := proc(t,R)loc(s,i)(
         forall(s=arg(t,i),
           nbt(s,R)
         );
         appls(t,R);
         t:=can(t)
);
```

Strategy applytb realizes a top-bottom strategy and repeats it when possible. Strategy applybt does the same but moves bottom-up.

Strategy ntr applies top-bottom rules when possible but makes one step up after each successful application.

```
NAME ntr;
ntr(t,R)(
          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)
);
```

Strategy lmt applies relations top-bottom until the first successive application and then continues from the very beginning. It may also be characterized as a leftmost outermost strategy.

```
NAMES lmt,lmtl;
lmt(t,R)(
  yes:=1;
  while(yes,lmtl(t,R))
);
lmtl := proc(t,R)loc(s,i)(
           t:=can(t);
           appls(t,R);
           yes → return;
```

```
        forall(s=arg(t,i),
          lmtl(s,R);
          yes → return
        );
        t:=can(t);
        return
);
```

Function can(t) calls the interpreter of the main operation of t and all strategies use this function so that after finishing, the working term will be represented in its main canonical form even if no rules from R were applied.

Strategies applytb, applybt and lmt are normalized, i.e. finishing the work only when no rules are applicable, and if the system R is canonical (confluent and noeterian), the call for strategy is invariant on strongly disjoint sets of states.

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 ω is introduced jointly with coefficient operation φ and two optional constants: neutral element e and annulator a. Except for associativity and commutativity, the following identities are true:

$$(x\varphi y)\omega(x\varphi z)=x\varphi(y+z);$$

$$x\omega e=x;$$

$$x\omega a=a;$$

$$x\varphi 0=e;$$

$$x\varphi 1=x;$$

$$e\varphi x=e;$$

$$a\varphi x=a.$$

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

$$x\omega x=x;$$

$$x\omega e=x;$$

$$x\omega a=a;$$

$$v(v(x))=x;$$

$$x\omega v(x)=o.$$

Information about ac-operations is collected in the data structure, which is the value of the standard name ac_list. This structure is an array of ac-descriptions.

Each description is 5-tuple. For arithmetical ac-operations the description is $((\ )\omega(\ ),(\ )\varphi(\ ),e,a,nil)$. For boolean-like operations the description is $((\ )\omega(\ ),v(\ ),e,a,o)$. If one of the 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)
             );
```

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 reduction of similar members for arithmetical operations and simplifications for both types of ac-operations. Function mrg and procedure merge reduce to canonical form expessions of the type $x\omega y$ where x and y are canonized and reduced to canonical form.

Function mrg may be used in rewriting systems, procedures merge and ord may be used for constructing strategies for the algebras with ac-operations. A useful example of such a strategy is can_ord. This strategy is equivalent to the following external procedure:

```
can_ord := proc(t,R1,R2)loc(s,i)(
             t:=can(t);
             appls(t,R1);
               forall(s=arg(t,i),
                 can_ord(s,R1,R2)
               );
             can_up(t,R2)
);
can_up := proc(t,R)loc(s,i)(
            appls(t,R);
            while(yes,
              forall(s=arg(t,i),
                can_up(s,R)
              );
              appls(t,R)
            );
            t:=can(t);
            merge(t)
);
```

Strategies for regular systems. Regular rewriting systems that are left-linear and nonoverlapping (no critical pairs) are of great importance in the theory and applications of rewriting technique. They are confluent but not necessarily noeterian. The completeness of strategies for regular systems means that they are terminated for any normalized term. It is known (O'Donnel) that the parallel outermost strategy is complete. This strategy may be presented by the following procedure in APS:

```
paraut := proc(t,R)loc(cont)(
  dowhile(
    cont:=applpar(t,R),
    cont
  )
);
applpar := proc(t,R)loc(s,i,cont)(
  applr(t,R);
  yes → return(1);
  cont:=0;
  forall(s=arg(t,i),
    cont:=cont || applr(s,R)
  );
  return(cont)
);
```

Strictly speaking this strategy is parallel outermost only if the system is right-linear. Otherwise, some subterms may be identified and additional reductions may appear at the next step reduction of outermost redexes. But for regular systems it may be proved that these additional reductions do not change the condition for R to be complete.

Huet and Levy [4] introduced the notion of needed redex occurrences and the strategy that reduces only needed occurrences was developed for a class of regular rewriting systems called strongly sequential. The notion of strong sequentiality, as well as the strategy based on this notion, depends only on the set of lhs's of rewriting systems. In [12] a nice generalization of the Huet–Levy theory was proposed based on the notion of strongly necessary sets of occurrences, and an algorithm was developed that finds minimal but in some sense strongly necessary sets and uses them for optimal reduction. In the special case of strongly sequential systems, this algorithm finds one of the needed occurrence and realizes the Huet–Levy strategy. The procedure nset presented below is a generalized version of the algorithm from [12]. It is based on a modification of applr.

The modified procedure appl(t,R) does the same as the original except that the name yes produces as the value of the standard name failset the set of occurrences which in the case when yes=0 satisfy the *completeness* condition w.r.t. t and the set L of lhs's of the system R. To formulate this condition, let us introduce the notion of the *compatibility* of the term t with the lhs l from the set L of lhs's. This notion is

recursive: t is *compatible* with l if it is an instance of l or there exist occurrences $p_1, \ldots, p_n$ such that $arg(t, p_1), \ldots, arg(t, p_n)$ are compatible with some lhs's from L and $t[p_1 \leftarrow t_1, \ldots, p_n \leftarrow t_n]$ is the instance of l for some $t_1, \ldots, t_n$.

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

By definition [12] the set Q of redexes is strongly necessary w.r.t. L if in an arbitrary reduction sequence by means of an arbitrary rewriting system with the set L of lhs's at least one of them or its residual is reduced.

Theorem 6.2. *Let $\{p_1, \ldots, p_k\}$ be complete w.r.t. t and $L, Q_1, \ldots, Q_k$ are correspondingly strongly necessary sets for $arg(t, p_1), \ldots, arg(t, p_k)$. Then if $Q_1 \cup \cdots \cup Q_k$ is not empty, this union is a strongly necessary set for t; otherwise, a nonempty strongly necessary set for any of* `arg(t,i)` *is strongly necessary for t.*

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

To be effective the modified `applr` must use some simple sufficient conditions for noncompatibility which might be checked simultaneously with matching. Such simple conditions exist for the so-called constructor systems that distinguish between defined and constructor operations: a term with a constructor operation at the root may never be compatible with any lhs. Exactly this kind of system is considered in [12] and our algorithm generalizes that approach to nonconstructor systems.

The strategy `nset` based on strongly necessary sets and Theorem 6.2 may now be represented as follows:

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

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

Every reduction that is made by one step of the algorithm, i.e. on the outermost call of applns, rewrites only redexes that belong to some strongly necessary set. It may be shown that this set includes the set generated by the Sekar–Ramakrishnan algorithm, and therefore for strongly sequential systems the unique strongly necessary occurrence is rewritten.

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

```
nset := proc(t, R, L)loc(cont, s, i)(
        dowhile(
          cont := applns(t, R, L),
        cont > 0);
        forall(s = arg(t, i),
          nset(s, R, L)
        )
);
applns := proc(t, R, L)loc(cont, cont1, l, q)(
            cont := 0;
            applr(t, R);
            forall(l in L,
              is_type(t, l) → (
              q → compat(t, l);
            equ(q, match) → (
              applr(t, R);
              return(2)
            )else q → arg(q, l);
            cont1 := applns(q, R, L);
```

```
            (contl == 2) → (
              applr(t, R);
              yes → return(2)
              else contl := 1
            );
            cont := cont || contl
        )
);
return(cont)
);
compat := proc(t, l)loc(p, q, i)(
  is_par(l) → return(match);
  p := match;
  is_type(t, l) → (
    for(i := 1, i <= ART(t), i := i + 1,
      q → compat(arg(t, i), arg(l, i));
      equ(p, match) → p - → q
    );
    return(p)
  );
  return(pt(t))
);
```

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

The proof of the correctness of the program applns is realized using induction on the depth of the term and the following invariant for the main loop. If t is the initial value of t and $l_1, \ldots, l_n$ are the lhs's already observed in the loop for all (l in $L, \ldots$) then there exists the set of occurrences of t which is complete w.r.t. t and $l_1, \ldots, l_n$ and the union of some necessary sets for these occurrences has already been reduced.

The procedures nset and applns allow some other improvements which eliminate repeated actions such as repeated observation of subterms which are compatible with no lhs's, but the main optimization may be obtained by means of mixed computations by substituting concrete rewriting systems into the strategy [9].

Strategy nset may also be applied to nonregular systems and after some modification to the systems with APS semantics (ordering, canonical forms and identifying of nodes), but it may lose the normalizing property and completeness. The repetition of the strategy may possibly make it normalizing, but completeness requires special

investigation. In practice, these problems are not very difficult and the strategy may be effectively used for the extended classes of the systems.

7. Data types

Multi-sorted algebras. The term algebra used in APS is one-sorted Ω-algebra generated by the set Z. Constants and data as well as names have no types, but types may be introduced and supported in many different ways. One of the most natural ways is to construct a multi-sorted algebra D using the subsets of the set $T=T_\Omega^*(Z)$. D is a family $D=(D_\xi)_{\xi\in\Xi}$ with the signature of types Ξ. All types in APS are realized by the subsets of the set T by the following construction. Let $D_\xi\subset T$, $\xi\in\Xi$ and for every operation $\omega\in\Omega$ the nonempty set $typeset(\omega)$ of admissible operation types, i.e. the expressions of the type $(\xi_1,\ldots,\xi_n,\xi)$ where $n=arity(\omega)$, $\xi_1,\ldots,\xi_n$, $\xi\in\Xi$ is given. If ω has more than one type this operation is polymorphic. The algebra T itself is one of the components of the algebra D, say D_τ, and so one of the admissible types is $(\tau,\ldots,\tau,\tau)$. The family D is called a free multi-sorted extension of the algebra T if the following closure conditions are satisfied: for every admissible type $(\xi_1,\ldots,\xi_n,\xi)$ for operation ω and for all $t_1\in D_{\xi_1},\ldots,t_n\in D_{\xi_n}$ the term $\omega(t_1,\ldots,t_n)\in D_\xi$.

The next step of construction will be factorization of the algebra D by some congruence relation which leaves D_τ as the free component. The algebra obtained in this way is called a multi-sorted extension of the data algebra T. The construction that was considered above may be generalized in several directions. Firstly, not only operations of the signature Ω but their superpositions may be considered as the operations of D. Secondly, the extension of source algebra may be constructed step by step, an already built and factorized multi-sorted algebra being considered as the material for new extension. The parameterization of types may be conveniently realized by inserting the type signature Ξ into the data algebra and defining the necessary operations on the types.

Type checking. The following example illustrates some possibilities for the realization of a multi-sorted extension with the tools of APLAN. Components of D are defined by means of predicates on T. Here is the piece of algebraic program.

```
NAMES check, subtype, checktype;
check := rs(t,s,x,y)(
                    (x;y)=check(x) & check(y),
                    (x,y:t)=check(x:t) & check(y:t),
                    (nil:t)=1,
      isname(x)→(x=check(vl(x))),
      isname(x)→((x:t)=check(vl(x):t)),
               ((x:s):t)=subtype(s,t),
               (x:(t of s))=check((x⇒s):t),
               (x:t)=t(x)
      );
```

```
subtype := rs(x)(
                    (x, any) = 1,
                    (x, x) = 1,
        (x of y, x of u) = subtype(y, u),
        x = 0
        );
checktype := rs(t, s, x, y)(
              (x; y) = (checktype(x); checktype(y)),
              (x, y : t) = (x ⇒ check(x : t), checktype(y : t)),
              (x : t) = (x ⇒ check(x : t)),
isname(x) → (x, y) = (x ⇒ check(x), checktype(y)),
isname(x) → (x) = (x ⇒ check(x))
);
```

The rewriting system check checks that the data belongs to the given type. A one-time application of this system to the list $(x, y, \ldots, z : t)$ transforms it to 1 if all data structures $x, y, \ldots, z$ belong to the type t. The types are presented by the names of recognizing predicates. Therefore, $t(x)$ is reduced to 1 or 0 dependently on the result of recognition. Operation of is used for parameterization of types: t *of* $(t_1, \ldots, t_n)$ defines the type depending on the parameters $t_1, \ldots, t_n$. The following statements define the type rsys of the rewriting systems of APLAN. The parameterized type list of(...) is used in this example.

```
NAMES list, par, eql, rsys;
list := rs(t, x, y)(
                  check(x : t) → (((x, y) ⇒ t) = list(t ⇒ y)),
                  check(x : t) → ((x ⇒ t) = 1)
        );
par := rs(x)(isname(x) ‖ isatom(x) → (x = 1), x = 0);
eql := rs(x, y, z)((x = y) = 1, (x → (y = z)) = 1, x = 0);
rsys := rs(x, y)(
          check(x : list of par; y : list of eql) → (rs(x)(y) = 1),
        x = 0
        );
```

Suppose now that the algebraic module which contains the above definitions is completed by the following statements:

```
typedef := (rsys : rdn, simpl, delmlt;
                list of par : plist);

task := prn(checktype(typedef));
```

Then if the values of names inserted in typedef meet the corresponding definitions, the procedure task will print:

```
rdn ⇒ 1, simpl ⇒ 1, delmlt ⇒ 1; plist ⇒ 1
```

Internal support for types. The computation of canonical forms must be generalized to work with multi-sorted algebras and data types. The program system CAN_ξ is used instead of CAN where ξ ranges over the signature of the types of some extension of the basic data algebra called canonical extension. Program CAN_ξ computes the function satisfying the relation

$$CAN_\xi(\omega(t_1, \ldots, t_n)) = \varphi_{\omega,\sigma}(CAN_{\xi_1}(t_1), \ldots, CAN_{\xi_n}(t_n)),$$

where $\sigma = type(\xi, \omega, t_1, \ldots, t_n) = (\xi_1, \ldots, \xi_n, \xi)$ is one of the admissible types for the operation ω. The function $CAN = CAN_v$ is used for the initial computation of the canonical form where v is the universal type. Functions $\varphi_{\omega,\sigma}$ are the interpreters of operations. Together with function type they define the canonical extension of the data algebra and the general algorithm to reduce the expressions to canonical form.

Function CAN now defines the system of equivalences

$$t = t'(CAN_\xi) \Leftrightarrow CAN_\xi(t) = CAN_\xi(t')$$

on the components D_ξ of canonical extension D of the data algebra which must determine the congruence relation on D. The sufficient condition for these equivalences to define a congruence is as follows: for every $\xi, \omega, t_1, \ldots, t_n$ there exists no more than one type $\sigma = (\xi_1, \ldots, \xi_n, \xi)$ such that $t_1 \in D_{\xi_1}, \ldots, t_n \in D_{\xi_n}$ and σ is admissible for ω. Use of canonical forms for multi-sorted algebras provides correct manipulation with application and quote operation. Indeed, let us consider the algebra with three sorts: T, T', F. T is absolutely free algebra, T' the algebra of terms, considered up to the main equivalence (equivalence defied by the main canonical form) and F the algebra of function descriptions. Then quote may be considered as the operation of the type (T, T), application has three admissible operation types (T', T', T'), (F, T', T'), (T, T, T), and the admissible types of other operations may be defined by the function type.

Another requirement to the realization of CAN is the protection of subobjects of the object t from being reduced to canonical form. Then applying CAN satisfies the invariance condition on the set of strongly disjoint states.

8. Concluding remarks

The devolvement of computational systems which integrate different paradigms of programming and support efficient programming tools presents difficult problems. To solve them one must use mathematical models for reasoning about programs, find clear conditions for their correctness and then prove it.

The procedural tools of APS are used first of all to write strategies of rewriting and enrich the rewriting systems by building into them different canonical forms on different levels of implementation. Proving the properties of such tools demands the use of clear semantics of programming language. This is especially important when graph rewriting is used instead of tree rewriting.

Formalisms that were described in this paper helped the authors to design the APS system and tools for its further devolvement.

References

[1] J.A. Bergstra, J. Hearing and P. Klint, eds., *Algebraic Specification* (ACM Press and Addison-Wesley, Reading, MA, 1989).

[2] M. Bidoit and C. Choppy, ASSPEGIQUE: an integrated environment for algebraic specifications, in: *Proc. Internat. Joint Conf. on Theory and Practice of Software Development* (Springer, Berlin, 1985), 246–260.

[3] J. Goguen, C. Kirchner, H. Kirchner, A. Megrelis and T. Wincler, An introduction to OBJ-3, in: J.-P. Jouannaud and S. Kaplan, eds., *Proc. 1st Internat. Workshop on Conditional Term Rewriting Systems* (Springer, Berlin, 1988).

[4] G. Huet and J.J. Levy, Computations in nonambiguous linear term rewriting systems, Tech. Report 359, INRIA, Le Chesney, France, 1979.

[5] J.-P. Jouannaud and S. Kaplan, eds., *Proc. 1st Internat. Workshop on Conditional Term Rewriting Systems* (Springer, Berlin, 1988).

[6] C. Kirchner and H. Kirchner, Reveur 3: implementation of a general completion procedure parametrized by built-in theories and strategies, *Sci. Comput. Programming* **20** (1986) 69–86.

[7] P. Lescanne, ed., *Rewriting Techniques and Applications*, Lecture Notes in Computer Science, Vol. 256 (Springer, Berlin, 1987).

[8] A.A. Letichevsky and J.V. Kapitonova, Algebraic programming in APS system, in: *Proc. ISSAC '90*, Tokyo, Japan (ACM, New York, 1990) 68–75.

[9] A.A. Letichevsky, J.V. Kapitonova and S.V. Konozenko, Optimization of algebraic programs, in: *Proc. ISSAC '91* (ACM Press, New York, 1991) 370–376.

[10] A.A. Letichevsky, J.V. Kapitonova and S.V. Konozenko, Algebraic programming system APS-1, in: O.M. Tammepuu, ed., *INFORMATICS '89, Proc. Soviet-French Symp.*, Tallinn (Institute of Cybernetics, Estonian Acad. of Sciences, 1989) 46–52.

[11] M.J. O'Donnell, Term rewriting implementation of equational logic programming, in: P. Lescanne, ed., *Rewriting Techniques and Applications*, Lecture Notes in Computer Science, Vol. 256 (Springer, Berlin, 1987) 1–12.

[12] R.C. Sekar and I.V. Ramakrishnan, Programming in equational logic: beyond strong sequentiality, in: *Proc. 5th Ann. IEEE Symp. on Logic in Computer Science* (IEEE Computer Society Press, Silver Spring, MD, 1990) 230–241.

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

[14] S. Wolfram, *Mathematica™. A System for Doing Mathematics by Computer* (Addison-Wesley, Reading, MA, 1988).