Computations in Extensions of Multisorted Algebras

Michael Lvov [0000-0002-0876-9928]

Kherson State University, 27, Universytets'ka St., 73000 Kherson, Ukraine

Lvov@ksu.ks.ua

Abstract. Development of algorithms of algebraic computations is one of the main problems, which arises with realization of mathematical software based on symbolic transformations. Multi-sorted algebraic systems (MAS) are mathematical model for this problem.

Present paper deals with the solution of this problem. We propose the approach to realization of interpreters of multi-sorted algebraic operations by its specifications, based on constructive improvement of notion of extension of multi-sorted algebraic system. This approach is illustrated by examples of realization of interpreters of operations in the field of rational numbers, ringing of one variable polynomial over the field, algebra of Boolean functions. Practice of this approach using for development of mathematical educational software shows its effectiveness and even universality.

Keywords. systems of computer mathematics, symbolic computations, multi-sorted algebras, extensions of algebras, interpreters of algebraic operations.

Introduction

Development of algorithms that perform algebraic computations is one of the main problems that arise when one implements mathematical systems based on symbolic transformations. A mathematical model of this problem is the notion of a multi-sorted algebraic system (MAS). The practice of development of simple educational mathematical systems [1, 2, 3] showed that implementation of algebraic computations requires careful preliminary design of MAS via development of MAS sort hierarchies and specifications of interpreters of multi-sorted algebraic operations [8]. Due to a number of reasons [9], for implementing calculations based on symbolic transformations we use the system of algebraic programming APS [4, 5, 6] which was adapted for our purposes by V. Peschanenko [10].

APS uses an algebraic programming technology based on rewriting systems and rewriting strategies. Thus, an interpreter of an algebraic operation is defined by a system of rewriting rules.

1 Problem formulation

In the paper we propose an approach to implementing interpreters of multi-sorted algebraic operations in accordance with their specifications which is based on the constructive refinement of the concept of an extension of a multi-sorted algebraic system. The definitions of a MAS and its (constructive) extension are given in paragraph 1. A typical example of a constructive extension is the example of the field of rational numbers as an extension of the ring of integers (example 2).

Constructive MAS extensions are classified as static, linear or binary dynamic extensions (definition 1.3.1, 2.1.).

We show that an interpreter of an algebraic operation in a constructive extension can be synthesized automatically in accordance with its specification which defines the rules of interpretation of the operation in extension, and the conditions of embedding of the base algebra in its extension.

The algorithm of synthesis of an interpreter of an operation is determined by the type of an extension. Therefore, in paragraph 2.2 we give examples of implementation of interpreters of operations of the field of rational numbers (static extensions), quadratic radicals field (binary dynamic extensions), univariate polynomial rings (linear dynamic extensions) and the algebra of propositions (binary dynamic extensions).

1.1 Multi-sorted algebras as mathematical model of algebraic computations

Definition 1.1. Let $U = \{u_1, \ldots, u_k\}$ be a finite set of symbols which is called the sorts signature. The symbols u_l, $l \in \{1,\ldots,k\}$ are called the names of sorts, or simply the sorts.

In particular, we will use the following sort names: *Variable*, *Bool*, *Nat*, *Int*, *Real*.

We will introduce other sort names within the definitions of the appropriate algebraic notions.

Definition 1.2. Let $\bar{S} = \{S_{u_1},\ldots,S_{u_k}\}$ be a finite family of sets indexed by sort names, the elements of which are called the value ranges of the corresponding sorts:

- $S_{Variable}$ is the set of variables,
- S_{Bool} is the set {False, True},
- S_{Nat} is the set of natural numbers,
- S_{Int} is the set of integers,
- S_{Real} is the set of real numbers.

Definition 1.3. A multi-sorted operation f on a family **S** is a map $f : S_{u_1} \times S_{u_2} \times \ldots \times S_{u_m} \to S_v$, where $u_1,\ldots,u_m,v \in \mathbf{U}$ are sorts of arguments and values of the operation f, respectively, and m is the arity of f.

The type of an operation is determined by the list of names of sorts of its arguments and the name of the sort of its range of values. The type of an operation f will be denoted as $(u_1,\ldots,u_m) \to v$. A signature Σ of operations is a finite set of symbols of operations together with a map that associates with each symbol $\varphi \in \Sigma$ a multi-

sorted operation f_φ together with its type (if φ is a symbol of an operation, then the expression $\varphi:(u_1,...,u_m)\to v$ that this symbol is associated with an operation of the type $(u_1,...,u_m)\to v$).

An example of a multi-sorted operation is scalar multiplication in a vector space. If *VectorSpace* is the sort name of a set of vectors over the field *Real* of real numbers, then the multiplication operation *Mult* "*" defines the map

$$Mult : Real \times VectorSpace \to VectorSpace$$

Below we will use more common, traditional mathematical notations for operations. Since the infix notation is usually used for scalar multiplication, we have:

$$Real * VectorSpace \to VectorSpace$$

Definition 1.4. Let *Bool* be a sort with the value range $S_{Bool}=\{True, False\}$. A multi-sorted predicate *P* is a mapping $P: S_{u_1}\times...\times S_{u_m}\to S_{Bool}$, where $u_1,...,u_m\in \mathbf{U}$, the sequence $u_1, ..., u_m$ determines the type of the predicate, and the number *m* is its arity. A signature Π of multi-sorted predicates is defined analogously to the signature of operations as a set of operations of predicate symbols, associated with multi-sorted predicates together with their types.

Definition 1.5. A multi-sorted algebraic system **A** is a tuple **A**=<**S, U,** Σ, Π >, where **S** is a set of sorts indexed by the symbols of the set **U**, $\Sigma=\{\varphi_1,...,\varphi_l\}$ is a signature of multi-sorted operations, Π = $\{\pi_1, ..., \pi_p\}$ is a signature of multi-sorted predicates.

Remark. Since the sort *Bool* can be added to the set of sorts, predicates can be considered as multi-sorted operations. Therefore, instead of considering multi-sorted algebraic systems, we will combine the signatures of operations and predicates and consider multi-sorted algebras.

Definition 1.6. Let **A**=<**S, U,** Σ > be a multi-sorted algebra and $u, v\in \mathbf{U}$ be sort symbols. We will say that the sort *v* depends on the sort *u*, if one of the operations of the signature Σ has the type of the form $u_1\times...\times u\times...\times u_m\to v$. As $\mathbf{U}_v$ denote a subset of sorts which depend on the sort *v*. Denote the subset of elements of Σ of type $u_1\times...\times u\times...\times u_m\to v$ as Σ_v, and the family of ranges of values of sorts $\mathbf{U}_v$ as $\mathbf{S}_v$. A restriction $\mathbf{A}_v$ of a multi-sorted algebra **A** to a sort *v* is the multi-sorted algebra $\mathbf{A}_v$=<$\mathbf{S}_v$, $\mathbf{U}_v$, Σ_v>.

Thus, a multi-sorted algebra **A** can be represented by a set of restrictions (algebras) $\mathbf{A}_v$, $v\in\mathbf{U}$, that $\mathbf{A}=<A_{u_1},...,A_{u_k}>$.

Example 1

Consider a software system that implements simplification of algebraic and trigonometric expressions. The core of the system must implement the computations in the ring of polynomials and the ring of multivariate trigonometric polynomials over the field of rational numbers. Specifications shall be given for the following algebras – restrictions to the mentioned sorts:

MultiPolynom – the ring of multivariate polynomials.
MultiPolynom + MultiPolynom → MultiPolynom
*MultiPolynom * MultiPolynom → MultiPolynom*
…
MultiTrig – the ring of multivariate trigonometric polynomials.
Sin(LinComb) → MultiTrig
Cos(LinComb) → MultiTrig
MultiTrig + MultiTrig → MultiTrig
…
LinComb – the vector space of linear combinations of several variables (arguments of trigonometric polynomials).
Pi
LinComb + LinComb → LinComb
*Rat * LinComb → LinComb*
…
Rat – the field of rational numbers (coefficients of polynomials and trigonometric polynomials).
Rat + Rat → Rat
Rat - Rat → Rat
Rat = Rat → Bool
Rat < Rat → Bool
…
The relation of dependence between sorts generates a structure of dependence on the set of algebras $\mathbf{A}_u$, $u \in \mathbf{U}$: an algebra $\mathbf{A}_v$ depends on the algebra $\mathbf{A}_u$ if the sort v depends on the sort u. If the relation of dependence has no cycles, then a multi-sorted algebra can be constructed step by step (incrementally) by constructing an algebra $\mathbf{A}_v$, if algebras $\mathbf{A}_u$ on which it depends are already constructed.

1.2 Axioms and constructs of multi-sorted algebra

For constructing algebras $\mathbf{A}_u$ we use their axiomatic and constructive descriptions (definitions).

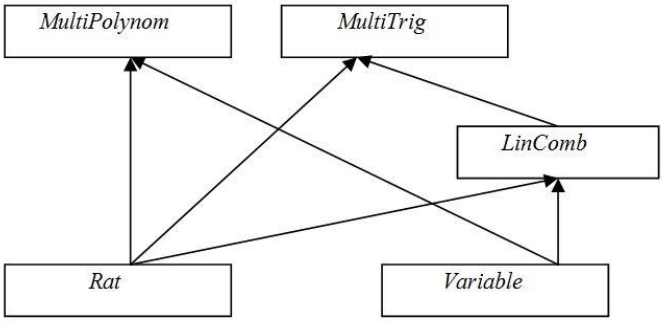

Fig. 1. Diagram of dependency of algebras of Example 1.

Definition 1.7. An axiom of an algebra $\mathbf{A}_u$ is an identity or a conditional identity in a signature $\boldsymbol{\Sigma}_u$. An axiomatic description in an algebra $\mathbf{A}_u$ is a finite set of axioms (axiom system) of the algebra $\mathbf{A}_u$.

We will use algebraic terminology and the relevant systems of axioms from the book [11]. A constructive description of an algebra $\mathbf{A}_u$ is a definition of a constructor of the sort $\mathbf{S}_u$ (i.e. a definition of terms of sort $\mathbf{S}_u$) and a set of interpreters of operations of $\boldsymbol{\Sigma}_u$.

Definition 1.8. A signature of constructors $\boldsymbol{T}_u$ is a finite set of symbols of operations together with a map that with each symbol $\tau \in \mathbf{T}_u$ associates a symbols of the sort u together with a list of symbols of sorts of its arguments (if τ an operation symbol, the expression $u = \tau(u_1,...,u_m)$ means that this symbol is associated with the symbol of the sort u and the symbols of the sorts of its arguments $u_1,...,u_m$.)

A constructor of sort $\mathbf{S}_u$ of an algebra $\mathbf{A}_u$ is a system of equations which defines syntactically the elements of sort $\mathbf{S}_u$ as terms in $\mathbf{T}_u$ signature. So, the sort $\mathbf{S}_u$ is the set of terms in (its own) signature $\mathbf{T}_u$ of constructors of $\mathbf{S}_u$ sort.

Definition 1.8 is the key one in our approach to specification of algebraic computations. Let us present the relevant examples and explanations.

Example 2. The field *Rat* of rational numbers.

Rational numbers are represented in the form of simple fractions. The constructor of a sort defines the standard representation of elements of this sort. The most common is the *canonical form*. That is why,

$$S_{Rat} = \{\frac{p}{q} : p \in S_{Int}, q \in S_{Nat}, GCD(p,q) = 1\} \tag{1}$$

Horizontal line is a symbol of the sort constructor. The same mathematical symbol is used to denote the operation of division, in particular in *Rat*. This is not convenient for the tasks of specification of algebraic computations. Therefore, we introduce the concept of a signature of operations Σ and a signature of constructors T. In particular, for denoting the constructor of the sort *Rat* we will use *double forward slash*:

$$S_{Rat} = \{p // q : p \in S_{Int}, q \in S_{Nat}, GCD(p,q) = 1\} \tag{2}$$

An important factor is that in the standard forms of presentation of elements of sorts the syntactic aspects of the definition are always combined with semantic aspects defined as contextual conditions i.e. predicates. In our case such a predicate is the equality *GCD(p, q)* = 1.

Example 3. The ring *Polynom* univariate polynomials over the field *Rat*.

Elements of this field are polynomials represented as sums of monomials, written in descending order of degrees. This definition should be recursive, and the concept of degree has to be defined separately.

$$\begin{aligned} &S_{Polynom} = \{Q : Q = M + +P, M \in S_{Monom}, P \in S_{Polynom}, \\ &\deg Q \overset{df}{=} \deg M; \deg(M) > \deg(P)\} \cup S_{Monom} \end{aligned} \tag{3}$$

To define the carriers of sorts we will use a special specification language, which allows non-recursive and recursive syntactic definitions of sorts' elements, definitions of the access functions and contextual conditions. For example:

```
Rat r ={(Int a)//(Nat b);    // Constructor of sort
  Num(r) = a, Den(r) = b;    // Access functions
  GCD(a, b) = 1      // Contextual condition
};
Monom M = {(Rat c)$(Const Variable x)^^(Nat n);
Coef(M) = c, Var(M) = x, Deg(M) = n // Access functions
};
Polynom P = {(Monom M)++(Polynom Q); // Constructor
LeadMon(P) = M,      // Access functions
LeadCoef(P) = Coef(M), Deg(P) = Deg(M);
Deg(P) > Deg(Q)        // Contextual condition
};
```

In order to implement computations in an algebra $\mathbf{A}_v$, v in $\mathbf{U}$, it is necessary to implement algorithms for performing each of its operations in such a way that the axioms of the algebra are satisfied.

Definition 1.9. An interpreter of an operation of a signature Σ_u is a function which is implemented by an algorithm which performs the corresponding operation.

Interpreters of operations are defined using a programming language. For our purposes we use *APLAN* language. So we include this language in the specification language.

Thus, for axiomatic and constructive description of an algebra $\mathbf{A}_v$ to its definition we add a finite set of axioms Ax_v and finite set of interpreters I_v. Then a multi-sorted algebra $\mathbf{A}_v$ is defined as follows: $\mathbf{A}_v = <\mathbf{S}_v, \mathbf{U}_v, T_v, \Sigma_v, Ax_v, I_v>$.

1.3 The methods of construction of multi-sorted algebras

Construction of the structure of multi-sorted algebras means specification, prototyping and implementation of algebraic computations. Specification of the structure of multi-sorted algebra is done in terms of extensions, homomorphisms, isomorphism and inheritance of multi-sorted algebras. Thus, together with the diagrams of dependence, which are graphical models of specifications of signatures of operations and constructors, the diagrams of extensions, diagrams of morphisms (isomorphism and homomorphism) and diagrams of inheritance are designed. We consider the method of extension. The methods of morphisms and inheritance are beyond the scope of this work.

1.3.1 The method of algebra extension

Definition 1.10. Let $\mathbf{A}_u$ and $\mathbf{A}_v$ be multi-sorted algebras. A multi-sorted algebra $\mathbf{A}_v$ is called an extension of $\mathbf{A}_u$, if $S_u \subseteq S_v$ and for any pair of operations f_1, f_2 of types

$f_1:(u_1,...,u_m)\to u$ and $f_2:(v_1,...,v_m)\to v$ respectively, if $S_{u_1}\subseteq S_{v_1},...,S_{u_m}\subseteq S_{v_m}$, then $\forall(a_1,...,a_m)\in S_{u_1}\times...\times S_{u_m}$ the equality $f_1(a_1,...,a_m)=f_2(a_1,...,a_m)$ holds.

An embedding is an isomorphic map $\mathrm{Re}d:\mathbf{S}_u\to\mathbf{S}'_v$, which maps $\mathbf{S}_u$ onto a subset $\mathbf{S}'_v\subset\mathbf{S}_v$. A restriction of an algebra A_v to a subset isomorphic to A_u, is determined by a system of conditional identities $E_1(x),...,E_k(x)$: $\mathbf{S}_v=\{a\in\mathbf{S}_v\mid E_1(a),...,E_k(a)\}$. The use of the system $E_1(x),...,E_k(x)$ as a rewriting system «simplifies» the term $a\in\mathbf{S}'_v$ to a term $a'\in\mathbf{S}_u:\mathrm{Re}d^{-1}(a)=a'$.

A constructive description of an extension $\mathbf{A}_v$ means a description of a constructor of $\mathbf{A}_v$ and an embedding of $\mathbf{A}_u$ into an algebra $\mathbf{A}_v$. In Figure 2 the double arrow indicates that $\mathbf{A}_v$ is an extension of the algebra $\mathbf{A}_u$, in $\mathbf{A}_v$ the construct $v=\tau(u_1,...,u,...,u_m)$ and embedding $Red_{u,v}$ are defined.

Note. The relation of MAS extension (the relation "sort-subsort") is basic in this work. Multi-sorted algebras, partially ordered by this relation, are called sorted-ordered. The fundamentals of the theory of sorted-ordered algebra in the applications to the theory of programming are presented in [12]. In Russian they are stated in [13].

Example 4. Consider the constructor of the field Rat (Example 2). According to the definition it defines a construct *Rat*, the arguments of which are sorts *Int* and *Nat*. Let us complement the specification of *Rat* sort by the embedding $\mathrm{Re}d:Rat\to Int$, defined by the equality $\mathrm{Re}d(a//1)=a$. Thus the sort *Rat* is defined constructively as an extension of sort *Int*.

Now consider the constructor of the *Polynom* ring (Example 3). It defines a recursive construct *Polynom*, the arguments of which is the sort *Monom*. Let us complement the specification of the sort *Polynom* with an embedding $\mathrm{Re}d:Polynom\to Monom$, defined by the equality $\mathrm{Re}d(M++0)=M$. So the sort *Polynom* is defined as an extension of the sort *Monom*.

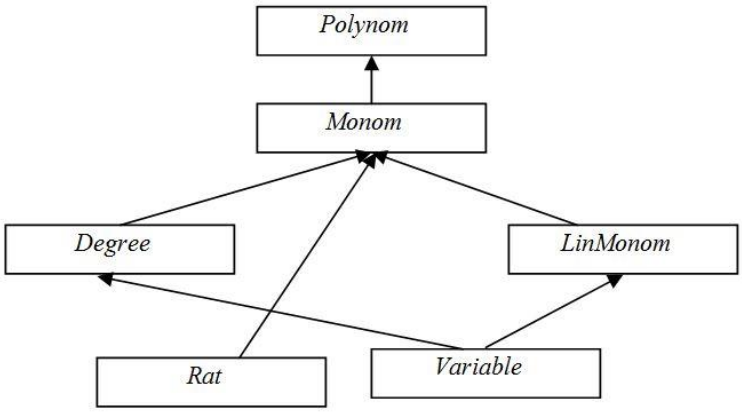

Fig. 2. Diagram of extensions in example 4.

In turn, the sort *Monom* is an extension of the sort *Degree* with a function *Red* defined by the equality **`1$x^^k = x^^k`**, extension of the sort *LinMonom* with a function *Red* defined by the equality **`a$x^^1 = a$x`** and extension of *Rat* with a function *Red* defined by the equality **`a$x^^0 = a`**. The Sorts *Degree* and *LinMonom* are extensions of the sort *Variable* with the functions of reduction, specified in accordance with the following equalities **`x^^1 = x`** and **`1$x = x`**. Thus, the extensions diagram has the following form

The extension mechanism is one of the main methods of specification of multisorted algebras. In particular, it allows defining the overloaded algebraic operations and algebraic type casting functions.

2 Methods of synthesis of algebraic programs

2.1 Static and dynamic extensions

Definition 2.1. An extension *B* of an algebra *A* is called static (non-recursive), if in its constructor $B = \varphi(A_1,...,A,...,A_n)$ none of the arguments coincides with *B*.

Examples of static extensions:

The field *Rat* is a static extension of the ring *Int*. Actually, $Rat\,R = (Int\,A)//(Nat\,B)$

The semigroup of monomials *Monom* with one generator is a static extension of the coefficients *Coef* field, because $MonomM = (Coef\,a)\$(Var\,x)\hat{}(Nat\,N)$.

Definition 2.2. An extension *B* of an algebra *A* is called dynamic (recursive), if in its constructor $B = \varphi(A_1,...,A,...,A_n)$ at least one of the arguments coincides with *B*.

A constructor of a dynamic extensions is a recursive definition, and therefore, contains both the base and recurrent part.

Examples of dynamic extensions:

A vector space *LinComb* of linear combinations of several variables over a field *Coef* is a linear dynamic extension of *LinMonom*, an element of which has the form **`a$x`**. An element $w \in LinComb$ has a form $w = a_1\$x_1 + +a_2\$x_2 + +...+ +a_m\$x_m$.

$LinComb\,w = (LinMonomu) + +(LinComb\,w)$

An univariate polynomial ring *Polynom* over the field *Coef*. This ring is traditionally denoted as *F*[*x*].

Polynom w = (Monom M)+ +(Polynom w)

Definition 2.3. A dynamic extension *B* of an algebra *A* is called linear, if in its constructor $B = \varphi(A_1,...,A,...,A_n)$ exactly one of the arguments coincides with *B*.

A dynamic extension *B* of an algebra *A* is called binary, if in its constructor $B = \varphi(A_1,...,A,...,A_n)$ exactly two of the arguments coincide with *B*.

Example 5. The field of square radicals

Examples 3 and 4 are examples of linear dynamic extensions. Consider an example of a dynamic binary extension:

The field *Rad*, the elements of which are linear combinations of square roots of square-free positive integers with rational coefficients, can be represented as a binary

extension of *Rad* using the following construction. Let $p_1,p_2,...,p_n...$ is a sequence of all prime numbers arranged in ascending order. Denote as *Q* the field of rational numbers. We introduce the following notations:

$$Rad_0 = Q,\ Rad_n = \{r : r = a + b * \sqrt{p_n},\ a,b \in Rad_{n-1}, n = 1,2,...\}.$$

The field *Rad* is the union of the increasing sequence Rad_n of fields.

$$Rad = \bigcup_{n=0}^{\infty} Rad_n,\ Rat = Rad_0 \subset Rad_1 \subset ... \subset Rad_n \subset ... \tag{4}$$

Thus, the constructor *Rad* has the form

$$Rad\, r = (Rad\, a) + (Rad\, b) * \sqrt{Nat\, p}\ |(Rat\ q) \tag{5}$$

Note that sequence of extensions (4) is a sequence of finite algebraic extensions of fields with roots of polynomials $x^2 - p_n = 0$.

Representation (5) includes a description of basic elements *Rat q* and a description of the extension mechanism - the constructor $(Rad\, a) + (Rad\, b) * \sqrt{Nat\, p}$. This specification exactly corresponds to the definition (4). On the other hand, the basic element description is unnecessary if it can be got from the embedding. Indeed, the specification of the *LinComb* vector space with the inclusion of the item describing of the elements of the basic algebra *LinMonom* has the form

$$LinComb\, w = (LinMonom\, u) + + (LinComb\, w)|(LinMonom\, u)$$

However, the inclusion $LinMonom \subset LinComb$ is defined by the equality *u* ++ 0=*u*. Therefore a separate description *LinMonom u* is unnecessary. We admit both types of specifications.

Formula (4) can be directly generalized to arbitrary dynamic extensions. If an algebra *B* is a dynamic extension of the algebra *A* with the constructor $B = \varphi(A_1,...,B,...,A_n)$, the increasing sequence $B_0 \subset B_1 \subset ..B_n \subset ...$ is defined as follows:

$$B_{(0)} = A, \tag{6}$$

$$B_{(n+1)} = \varphi(A_1,...,B_{(n)},...,A_n) \tag{7}$$

The embedding $\mathrm{Re}d : A \to B$ defines the embedding $\mathrm{Re}d_i : B_{i+1} \to B_i$, from which one obtains a representation of A in the form of an increasing sequence of algebras, where each one is a static extension of the previous one.

$$B = \bigcup_{n=0}^{\infty} B_n,\ B_0 \subset B_1 \subset ... \subset B_n \subset ... \tag{8}$$

In MAS development practice there were some generalizations of the definition (8). Namely, instead of the sequence of algebras $\{B_i\}_{i=0}^{\infty}$ let consider the set of indexed algebras $\{A_i\}_{i \in I}$, where *I* is a linearly ordered set of indices. An algebra $B_J, J \subset I, |J| < \infty$

is defined as a union of algebras A_j, $j \in J$: $B_J = \bigcup_{j \in J} A_j$. Let us assume there is an embedding of algebras B_j, where $B_{J_1} \cup B_{J_2} \subset B_{J_1 \cup J_2}$. Then

$$B = \bigcup_{J \subset I, |J| < \infty} B_J \tag{9}$$

An example of such algebra is a ring $K[[x]]$, the elements of which are sums of monomial with rational degrees:

$$K[[x]] = \{P : P = \sum_{j \in J, J \subset Rat, |J| < \infty} a_j x^j \tag{10}$$

Dynamic extensions are sequences of static extensions. This allows one to use the general scheme of implementation of dynamic extensions to derive the appropriate rewriting systems.

2.2 Synthesis of algebraic programs

2.2.1. Example of an algebraic program output with sort specifications

Example 7. Below is a specification of *Rat* sort and a derivation of calculations with rational numbers. Specifications of the *Rat* sort determine this sort as a field, linear order and static extension of *Int*.

```
Sort Rat:: Field, LinOrd;     //Inherited
Constructor
Rat r ={(Int a)//(Nat b); // Sort constructor
  a//1 = a;       // The embedding function  RatToInt
  Num(r) = a, Den(r) = b; // Access functions
  GCD(a, b) = 1    // Contextual condition
  Form: Num(Form(r)) ∈ Int, Den(Form(r)) ∈ Nat,
  GCD(Num(Form(r), Den(Form(r)) = 1;
};
Operations
Add:  a//b + c//d = Form((a*d + b*c)//(b*d));
Sub:  a//b - c//d = Form((a*d - b*c)//(b*d));
Mult: a//b * c//d = Form((a*c)//(b*d));
Div:  a//b / c//d = Form((a*d)//(b*c));
Div:  a/b = Form(a//b),a/0 = Exeption('Divison by zero');
Pow:  n >= 0 -> (a//b)^n  = (a^n//b^n),
n < 0 ->   (a//b)^n  = (b^-n//a^-n);
Predicates
Equ:  a//b == c//d = (a == c)&(b == d);
Gre:  a//b > c//d = (a*d > b*c);
Les:  a//b < c//d = (a*d < b*c);
UnLes: a//b >= c//d = (a//b > c//d) ∨ ( a//b == c//d);
UnGre: a//b <= c//d = (a//b < c//d) ∨ ( a//b == c//d);
```

Consider the output of the *Add* operation interpreter. From specifications we have:

```
a//b + c//d = Form((a*d + b*c)//(b*d));        (11)
```

```
a//1 = a;                                      (12)
```

we get:

```
a + c//d = Form((a*d + 1*c)//(1*d));           (13)
```

Applying the sort equality `Int  a*1 = 1*a = a` to (13), we obtain:

```
a + c//d = Form((a*d + c)//d)                  (14)
```

Similarly, for the second operand:

```
a//b + c = Form((a + b*c)//b);
```

The resulting relations are particular cases of (12) - a specification of the Add operation with the RatToInt embedding. Together with the general relation (11) they define implementation rules for addition of fractions:

```
Add:=rs{
a//b + c//d = Form((a*d + b*c),(b*d)),
a + c//d = (a*d + c)//d,
a//b + c = (a + b*c)//b
};
```

Similarly interpreters of operations of subtraction and multiplication on *Rat* can be derived. An exception is the division operation, which is absent in the signature of *Int* sort. Therefore, it is necessary to define and specify it as multi-sorted (see specification of the *Rat* sort). Another exception is the exponentiation operation, which is expressed using multiplication.

Now let us focus on the *Form* function. Definition of this function connects it with the symbol of sort constructor. The role of this function is essentially in the canonization of sort element. During execution of a rule of the general form this function is called on the result of the operation. Therefore, *Form* is an interpreter of the constructor sort symbol. For the symbol // of *Rat* sort constructor we will use the notation !/. The general rule of addition will take the form

```
a//b + c//d = a*d + b*c)!/(b*d).
```

We will always include the sign "!" in the infix notation of sorts constructors and it will always mean the call of the constructor of sort interpreter.

```
Add:=rs{
 a//b + c//d = (a*d + b*c)!/(b*d),
 a + c//d = (a*d + c)//d,
 a//b + c = (a + b*c)//b
};
```

Interpreter of sort constructor is called only in the first rule.

The method of derivation of an interpreter of an operation of sort *v* from its specification in the case when the algebra A_v is defined as a static extension of algebra A_u can be generalized as an algorithm of synthesis of an algebraic program.

Note that this method can be implemented in the form of an algebraic program because it relies only on equational derivation. For automation of elimination of the *Form* functions more sophisticated methods and technologies have to be used, e.g. a theorem prover over the basic sort *u*.

2.2.2 Interpreters output in linear dynamic extensions

Example 8. Specifications of *Polynom* sort and calculations with univariate polynomials.

Specifications of the *Polynom* sort define this sort as a Euclidean domain and a linear dynamic extension of *Monom*.

```
Sort Polynom::EuclideDomain;
Parameter Field Coef, Const Variable Argument;
Constructor{
Polynom P = Monom M ++ Polynom Q   // Constructor of sort
0 ++ P = P,         //Embedding function  PolynomToPolynom
M ++ 0 = M;         //Embedding function  PolynomToMonom
LeadMon(P) = M,              // Access functions
LeadCoef(P)= Cf(M),
Arg(P)= Arg(M), Deg(P)= Deg(M);
Deg(M)> Deg(Q), Arg(M)= Arg(Q); //Contextual condition
Form: M∈Monom, Q ∈ Polynom,
 0 ++ P = P, M ++ 0 = M,
 Arg(M) = Arg(Q), Deg(M)> Deg(Q), Cf(M)<>0
};
Operations
Add: Deg(a)==Deg(b)→ (a++A)+(b++B)=(a+b) !+ (A+B);      (15)
Sub: Deg(a)==Deg(b)→ (a++A)-(b++B)=(a-b) !+ (A-B);
Mult:   // Polynom * Polynom → Polynom; Commutative
(a++A)*(b++B)=(a*b)++((a*B+A*b)+A*B);
Mult:     // Coef * Polynom → Polynom; Commutative
 c*(b++B)= c*b ++ c*B;
 (b++B)*c = Form(c*b, c*B);
Div: (a++A)/b = a/b ++ A/b;
Pow: a^n=sqr(a^n div 2)*a^(n mod 2); // From sort
MiltSemiGroup
IntDiv:
 Deg(P)==Deg(Q) → P div Q = LeadCoef(P)/LeadCoef(Q),
 Deg(P)< Deg(Q) → P div Q = 0,
 Deg(P)> Deg(Q) → P div Q = LeadMon(P)div LeadMon(Q)++
 P-(LeadMon(P)div LeadMon(Q))*Q div Q);
Mod: P mod Q = P - (P div Q)*Q; // From EuclideDomain
```

In this example, we show that the methods of derivation of specifications discussed above leads to mathematically sound systems of interpretation rules.

First of all, note that there are two fundamentally different methods of definition operations. Operations *Add, Sub, Mult, Div* are defined in terms of constructors of operands. Such a definition of operation will be called constructive. It demonstrated by the definition of the operation *IntDiv* (division with remainder). *Mod* operation is defined in terms of operations of signatures *Polynom* sort. We will call such definitions of operations abstract or derived. Since this signature is inherited from the abstract sort *EuclideDomain*, the specification of *Mod* is given in this sort. In *EuclideDomain* sort Euclidean algorithm is defined. *Pow* operation should be defined earlier – in specification of the sort *MiltSemiGroup*.

A constructor of a sort is defined recursively. So the *Polynom* algebra is a sequence of nested algebras which begins with the algebra *Monom* (Monoms of one variable):

$$Mon = Pol_0 \subset Pol_1 \subset ... \subset Pol_k \subset ... \tag{16}$$

The algebra Pol_i is a set of polynomials of i-th degree. Then Pol_i is a vector space of dimension $i + 1$. In this interpretation polynomial degree determines the index in the sequence. Therefore, the operation of addition *Add* (15) is determined by three equalities, the first of which defines the rule of addition, if both operands belong to one algebra, the other two – when they belong to different algebras:

$$a,b \in Pol_i; a \in Pol_i, b \in Pol_j, i < j;\ a \in Pol_i, b \in Pol_j, i > j$$

Thus, the extension (16) is an extension of vector spaces. The rules of interpretation of vector operations are derived from their specifications quite similar to the case of static extensions:

$$\dim(Pol_i) = i+1,\ \dim(Pol_{i+1}) = i+2,\ Pol_i \subset Pol_{i+1},$$

$$V \in Pol_{i+1} \to V = a{+}{+}A, A \in Pol_i; 0{+}{+}A = A,\ a{+}{+}0 = a.$$

```
Deg(a)==Deg(b)→(a++A)+(b++B)=(a+b)!+(A+B),  //basic rule
Deg(A)< Deg(b) → A + (b++B)= b !+ (A+B),  //partial cases
Deg(a)> Deg(B) → (a++A) + B = a !+ (A+B)
```

The derived rules still do not consider the second of the conditions **M++0=M**. Therefore, each of these rules should still be converted:

```
Deg(a)== Deg(b)→ a + (b++B)=(a+b)!+ B.
Deg(a)== Deg(b)→(a++A) + b=(a+b)!+ A.
```

So, for the *Add* operation of sort *Polynom* we obtain the following system of rules:

```
Deg(a)== Deg(b)→ (a++A) + (b++B) = (a+b)!+ (A+B),
Deg(a)== Deg(b)→ a + (b++B)=(a+b) !+ B,
Deg(a)== Deg(b)→ (a++A) + b = (a+b) !+ A,
Deg(A)< Deg(b) → A + (b++B) = b !+ (A+B),
Deg(A)< Deg(b) → A + b = b ++ A,
Deg(a)> Deg(B) → (a++A) + B = a !+ (A+B),
Deg(a)> Deg(B) → a + B = a ++ B;
```

Also on the sort *Monom* we define an additional partial operation *Add*:

```
a$x + b$x = (a + b)$x.
```

This system takes into account both conditions, i.e. of a sequence of extensions

$$Monom = Pol_0 \subset Pol_1, \; Pol_i \subset Pol_{i+1}$$

Conclusion. Specifications of the sort *Polynom* determine a dynamic extension of the vector space. Derived operations should be excluded from the specifications of *Polynom* and included in specifications of the relevant abstracted algebras. Constructive operations of the signature of vector space are defined by the main case. The special cases are derived by the methods of derivation of static extensions.

Because there are two embedding relations for the sort *Polynom*, the derivation of a complete rules system is done sequentially: firstly by the first relation and then by the second one. Multiplication and an incomplete division are specified separately as additional operations on the vector space *Polynom*.

2.2.3 Example of algebraic program output within sort specifications. Dynamic binary extension

Consider an example of a binary dynamic extension of the Bool algebra – the algebra of logic *BoolAlg* [14]. This algebra is an extension of the *Variable* base sort, because the elements of sort are Latin letters interpreted as logical formulas. We will show that derivation of interpreters of logical operations is performed by the same methods.

The elements of the set *BoolAlg* are formulas of the propositional logic of many variables. Let $F(x_1, x_2, \ldots, x_n)$ be an arbitrary formula of propositional logic of n variables. Denote as O and I the truth and falsity values respectively. Then

$$F(x_1, x_2, \ldots, x_n) = x_n \,\&\, F(x_1 x_2, \ldots, x_{n-1}, I) \vee \neg x_n \,\&\, F(x_1, x_2, \ldots, x_{n-1}, O).$$

If we denote

$$A(x_1, \ldots, x_{n-1}) = F(x_1, \ldots, x_{n-1}, I), B(x_1, \ldots, x_{n-1}) = F(x_1, \ldots, x_{n-1}, O),$$

we obtain the representation

$$F(x_1, x_2, \ldots, x_n) = x_1 \,\&\, A(x_1, \ldots, x_{n-1}) \vee \neg x_1 \,\&\, B(x_1, \ldots, x_{n-1}) \qquad (17)$$

Now perform sequentially the same transformations of formulas A, B w.r.t. the variables $x_{n-1}, \ldots, x_n$. As a result we obtain a recursive representation of the propositional logic formulas. Indeed, through $BoolAlg_m$ denote the set of propositional logic formulas in variables $x_1, \ldots, x_m$. Then

$$BoolAlg_n = \{F : F = x_n \,\&\, A \vee \neg x_n \,\&\, B, A, B \in BoolAlg_{n-1}\}$$

Thus, sort *BoolAlg* is the union of an increasing sequence of algebras $BoolAlg_m$:

$$BoolAlg_0 = Bool, \; BoolAlg_0 \subset BoolAlg_1 \subset \ldots \subset BoolAlg_m \subset \ldots \qquad (18)$$

$$BoolAlg = \cup BoolAlg_m$$

Note that formula (17) defines a canonical form of a formula of the algebra of propositions. Denote

$$BF(A,B,x) \stackrel{df}{=} x \& A \vee \neg x \& B \tag{19}$$

Then

$$BF(A_1,B_1,x) \& BF(A_2,B_2,x) = BF(A_1 \& A_2, B_1 \& B_2, x), \tag{20}$$

$$BF(A_1,B_1,x) \vee BF(A_2,B_2,x) = BF(A_1 \vee A_2, B_1 \vee B_2, x) \tag{21}$$

$$\neg BF(A,B,x) = BF(\neg A, \neg B, x). \tag{22}$$

Thus, the basic logic operations are performed per the argument! Finally, it is easy to check that the embedding function is defined by the equality

$$BF(A,A,x) = A) \tag{23}$$

Conclusion

In this paper we have shown that the concept of a constructive extension of MAS is the key one in design and implementation of symbolic computations. Most of all, this applies to symbolic computations in mathematical systems for educational purposes, where classical algebras and algebraic systems are used.

Actually, the constructive approach, along with the axiomatic approach in algebra is well known [12-13]. The idea of a constructive definition of an algebra element through the elements of basic algebras is systematically used in algebraic research. On the other hand, the overloading mechanism for algebraic operations is a standard tool in programming of mathematical systems.

Thus, the main theoretical result is the idea of systematic usage of the construct of extension in programming of MAS signatures as overloaded signatures. Other methodical components of proposed approach are given in [14-17].

The practice of usage of this approach in development of mathematical systems for education has shown its effectiveness and even universality. This is the main practical result.

References

1. Lvov, M., Kuprienko, A., Volkov, V.: Applied Computer Support of Mathematical Training. Proceedings of Internal Workshop in Computer Algebra Applications. 25-26. Kiev. (1993).
2. Lvov, M.: AIST: Applied Computer Algebra System. Proceedings of ICCTE'93. 25-26. Kiev. (1993).

3. Lvov, M.: Therm VII - school computer algebra system. Computer in school and family.7, 27-30 (2004).
4. Letichevsky, A., Kapitonova, J., Volkov, V., Chugajenko, A., Chomenko, V.: Algebraic programming system APS (user manual). Glushkov Institute of Cybernetics, National Academy of Sciences of Ukraine, Kiev (1998).
5. Kapitonova, J., Letichevsky, A., Volkov, V.: Deductive means of the system of algebraic programming. Cybernetics and system analysis. 1, 17–35 (2000).
6. Kapitonova, J., Letichevsky, A., Lvov, M., Volkov, V.: Tools for solving problems in the scope of algebraic programming. Lectures Notes in Computer Sciences. 958, 31-46 (1995).
7. Lvov, M.: Basic principles of constructing pedagogical software to support practical classes. Control systems and machines. 6, 70-75 (2006).
8. Peschanenko, V.: Extending standard modules algebraic programming system APS for use in systems for educational purposes. Scientific journal of the National Pedagogical University named after M.P. Drahomanov. 3 (10), 206-215 (2005).
9. Peschanenko, V.: About one approach to the design of algebraic data types. Problems of programming (Problemy prohramuvannia). 2-3, 626-634 (2006).
10. Peschanenko ,V.: Use of the algebraic programming system APS to build systems for supporting the study of algebra in school. Upravlyayuschie sistemy i mashiny (Control systems and machines). 4, 86-94 (2006).
11. Van der Varden, B: Algebra. Nauka, Moscow (1979).
12. Goguen, J., Meseguer, J.: Ordered-Sorted Algebra I: Partial and Overloaded Operations. Errors and Inheritance. Technical Report, SRI International, Computer Science Lab (June 1983).
13. Goguen J.A., Meseguer J. (1987) Models and equality for logical programming. In: Ehrig H., Kowalski R., Levi G., Montanari U. (eds) TAPSOFT '87. TAPSOFT 1987. Lecture Notes in Computer Science, vol 250. Springer, Berlin, Heidelberg
14. Lvov, M.: About one approach to the implementation of algebraic calculations: calculations in the algebra of statements. Bulletin of Kharkiv National University (Series “Mathematical Modeling. Information Technology. Automated Control Systems”). 863, 157-168 (2009).
15. Lvov, M.: About one approach to verification of algebraic calculations. Problems of programming. 4, 23-35 (2011).
16. Lvov, M.: Inheritance method for the implementation of algebraic calculations in mathematical systems of educational purpose. Systems of control, navigation and communication. 3(11), 120-130 (2009).
17. Lvov, M.: The method of morphisms of the implementation of algebraic calculations in mathematical systems of educational purpose. Information processing systems. 6(80), 183-190 (2009).