INTERRELATION OF FORMAL COMPONENTS PROGRAMMING MACHINE

The formal components of the TL system can be considered in various combinations as higher-level formalisms. The connection between the components can be seen at the level of the structure of the construction of elements of the carrier sets of the algebraic systems discussed above. Here is a description of the main relationships.

Relationship between algebras M and K.

There is a mapping from the set M to the set K, in which K is the domain of definition, i.e. Dom(M)=K. Cartesian product M×K represents the space of possible states of the RAM Q when executing programs compiled using the IS L. A more general case is the introduction of multi-sorting for the sets M and K:

$$Q=M1\times K1\cup M2\times K2\cup ...\cup Mn\times Kn;$$

where Q is a universe consisting of pairs of the form $(m_i^t,\ k_j^t) \in M_t \times K_t$ with the sort t, which, for example, represents some type of variables and constants, and the symbol $\cup$- set-theoretic operation of union.

Selected subsets Q, among the elements of which there are no pairs with the same first member, represent computer memory states on the set of which predicates of pre- and postconditions are considered when proving the correctness of the program.

Relationship between algebras A and F

Both of these sets describe the syntactic constructions of the language L, while the union of the sets A and F describe the complete set of syntactic constructions L. According to the definitions made earlier for A and F, these algebraic systems do not define the language itself, but only list its functional chains indicating the type (number constructions) and locality (number of occurrences of non-terminal symbols). For these algebraic systems, it is possible to define a general multisorted algebra, the carrier of which will be the union of A and F, and the operations will be the corresponding union of the operations of these algebras.

Relationship between algebras I and Φ

Similar to algebraic systems A and F, systems I and Φ also have functions of the same type and can be combined into a single multi-sorted algebra.

Communication between systems Q, A∪F,I∪F

The connection between these PM components is determined mainly at the level of carrier sets. The most important relationships are the following:

$Dom(A\cup F) = M$, $Ran(A) = I$, $Ran(F) = \Phi$,

$Ran(A\cup F) = I\cup F$, $Dom (I\cup\Phi) = Q$,

where Dom and Ran mean, respectively, the domain of definition and the range of values of the set of elements indicated in brackets. The concepts of domain of meaning and domain of definition in this case are not strictly formal, since they reflect only the general meaning of the interaction of algebraic systems. More

precisely, these relations exist between the sets that support the corresponding algebras.

The choice of algebraic systems that will be used to analyze constructions from L is determined by the problem for which the algebraic formalism is supposed to be applied. For example, when studying language constructs to optimize programs and study their equivalence, in most cases it is sufficient to consider systems M, F, F; to prove the correctness of programs - Q, A, F; to justify the functional integrity of PL constructs, systems F, A, F are needed ,I and M.

To study the processes occurring during the execution of a specific program in the language L, it is assumed that instead of some algebraic variables used in the constructions of $A \cup F$, the constants of the set K will be used.

In this case, the constants are classified as syntax constructions with zero locality (fik) = 0, along with variables corresponding to elements from M, and they complement the set AU F. In this case, the relation is satisfied

$$Dom(S) = Q, \text{ where } S = A \cup F \cup K \cup M.$$

A set of sets P = <Q, S, R>, in which R are relations on Q, is an algebraic system P, in which Q, as a set of states of RAM, is the carrier, and S is a set of all possible operations (programs), acting as single functions. It should be noted that in the general case, the operations of composition of terms in this algebraic system can be partial, i.e. not defined everywhere, since not all syntactic constructions can be substituted into others in place of variables.

Thus, for qp, $qe \in Q$ and $sn \in S$, the relation sn(qp)=qe can be fulfilled, which should be interpreted as “the program sn transfers the state of the RAM qp to the state qe as a result of its execution.”

Subset of Qp states$\subseteq$Q is called the complete domain of the program sn if for each of the elements $qp \in Qp$ program sn guarantees its corresponding mapping into any element $qe \in Qe$, where $Qe \subseteq Q$. The set Qe is called the complete range of values of the program.

In practice, in most cases, some subsets of Qp* are considered$\subseteq$Qp and $Qe^* \subseteq Qe$, carrying the most informative fragment of the mapping sn, necessary to perform the target function of the program formulated by the programmer.

Logical predicates Pp and Pe that take the value “true” on the domains of definitions Qp* and Qe* are usually called pre- and postconditions of the program sn, respectively. Moreover, for any postcondition Pe' of interest to the program researcher, one of the important concepts is the weakest precondition Pp' for this postcondition. It is defined as the weakest condition to which it is sufficient to subordinate the initial state in order for the execution of the program sn to terminate and produce a state satisfying Pe'.

Algebraic properties of program machine elements

The elements of the software machine of any tool are algorithmic algebraic systems. Each of these systems has its own carrier set, signature, and set of

relationships. In this case, the properties of the algebraic system are the properties of signature operations and the properties of relations.

The set of properties of the software machine completely determines the specifics of the software tool. For example, algorithmic languages that have isomorphic programming engines are functionally equivalent. We call systems of mappings of program machines into each other homomorphisms of program machines. Let's take a closer look at this concept.

Let two program machines Tx and Ty be given, having the following components:

$$T_x = < M_x, K_x, A_x, F_x, \Phi_x , I_x > ,$$

$$T_y = < M_y, K_y, A_y, F_y, \Phi_y , I_y > .$$

We will call the mapping from Tx to Ty a homomorphism$\alpha$and designateα: Tx$\rightarrow$Ty, if the corresponding algebraic systems of these software machines also have homomorphic maps:

Mx$\rightarrow$My, Kx$\rightarrow$Ky, Ax$\rightarrow$Ay, Fx$\rightarrow$Fy, Fx$\rightarrow$Fu, Ix$\rightarrow$Iy.

For displaysαThe following generally accepted concepts are also used:

domain α:Dom(α) = A,

range α: Ran(α) = B,

imageα: Im(α) = {α(a)$\mid$a$\in$A},

α- *injection*:$\forall$a1, a2$\in$A[α(a1) =α(a2) => a1 = a2],

α- *surjection*: Im(α) = Ran(α),

α- *bijection*: simultaneouslyα- injection andα- surjection.

That is, an injection presupposes some one-to-one mapping for some elements of the domain of definition, a surjection speaks of a complete domain of definition, and a bijection speaks of a one-to-one mapping.

If we consider any two corresponding algebras X and Y of two software machines, then the following definitions can be made. Let algebras have operations of the same type {σ_1, ...,σ_n}. Then a homomorphism of algebras is the mappingα: X0$\rightarrow$Y0, where X0 and Y0 are sets that support algebras, under which the following statement holds:

$\forall$ σ_i:α(σ_i(x1, ..., xm)) =σ_i(α(x1), ...,α(xm)) ,

where { x1, ..., xm } = X0,α(xi) = yi, yi$\in$Y0.

If

α- injection, then it is called a monomorphism of algebras,

$\alpha$is a surjection, then it is called an epimorphism,

$\alpha$is a bijection, then it is called an algebra isomorphism.

We will say that there is a monomorphism between two software machines if there is a monomorphism between all their corresponding algebras. Similarly, machines are in epimorphism if the same mapping exists for their corresponding algebras. The strongest mapping of programmable machines is isomorphism, which is also defined through isomorphisms of the constituent algebras.

Note that the listed morphisms quite accurately make it possible to determine not only the relations between the sets-carriers of the algebras of software

machines, but also the relations between the properties of their operations. This follows from the fact that any algebra of program machines generates its carrier set using signature operations from a finite set of elements - the basis.

The study of morphisms of software machines allows us to identify various classes and subclasses of the corresponding tools. If isomorphism indicates the equal power and equivalence of the functional capabilities of algorithmic languages, then epimorphism may indicate that one of the languages is a functional subset of another. Epimorphism usually occurs for two tools of different versions. The earlier version of the tool must have an injective mapping to the later one. If this property is violated, software compatibility is violated, which is a gross violation of the discipline of designing software systems.

At the same time, in most cases, for different software machines, if they do not correspond to the same type of algorithmic languages (for example, universal languages), it is impossible to find a direct mapping of their designs into the designs of other tools. In this case, one can search for subsets of these means for which any of the homomorphisms turns out to be valid. Such a comparison of tools is necessary because it makes it possible to formally evaluate its advantages and disadvantages in comparison with others previously known. In addition, if a weak point is found in the toolkit, it can be eliminated if there is an understanding of the reason for its occurrence.

Software machine homomorphisms can also be used to examine the conceptual integrity of a tool by comparing it with earlier versions or other established tools.

The algebras that make up the programming machine use mainly the operations of composition of elements, for example, the composition of memory elements or the composition of operator constructions. Relations that complement algebra to an algebraic system allow us to consider the properties of these compositions. The need to study the properties of composition operations is determined by the following factors:

- when analyzing compositions, it becomes possible to replace some operations or operators with others, making the program more efficient,
- equivalent transformations based on the compositional properties of a software machine can lead to the concept of structural design with increased capabilities for debugging programs,
- tool designers can use the composition properties of tool elements to transform it into more flexible versions (for example, adaptive software),
- performing compositions on elementary software structures is the main way of designing software systems, and therefore contributes to the development of design methodology.

Some changes that a designer can make to his toolkit, knowing the basic principles of studying software machines, lead to the emergence of new compositional properties of the toolkit, making it more attractive to the user. The strongest compositional properties are possessed by those software tools that, based on their properties, can be classified as well-studied universal types of algebras. These algebras have their own names and established methodology. Such universal

algebras include groups, rings, lattices, fields and various types of their varieties [120,121,122]. The correlation of elements of a program machine to certain known types of algebras will be called identification of a program machine.

Before considering some classes of universal algebras that can be useful in identifying program machines, we present the basic properties of operations that occur in various universal algebras.

Operation$\sigma$is called unary if it has one argument, i.e.$\sigma: X \to X$, or mapping the domain into itself. An operation is called binary if it has two arguments, i.e.$\sigma: X \times X \to X$, where$\times$- Cartesian product of sets, operations that implement the mapping $X \times X \times X \to X$ are called ternary, etc. . For our purpose, it will be sufficient to consider the properties of some binary operations, since they are the most common case for elements of program machines.

Let there be binary (two-argument) operations in some algebraσAndγ. Then the operationσwill be called idempotent if for any element $x \in X$ statement is true$\sigma($ x, x) = x. In some network programming languages, operations of parallel composition of alternative program branches are idempotent. In this case, the composition of two identical branches will be functionally equivalent to one such branch. The same can be said about alternative composition in parallel programs [87,88,89,123].

Operation$\sigma$is called commutative if the statement is true for it$\sigma(x1, x2) = \sigma($ x2 , x1) , x1 , $x2 \in X$. For example, the operation of alternative composition in parallel program machines, or the operation of sequential composition of two operators independent of each other, can be commutative.

Operationσwill be called associative if the statement is true for it$\sigma(x1, \sigma($ x2 , x3)) $= \sigma(\sigma($ x1 , x2), x3) , x1, x2,$x3 \in X$. Most compositional operations of software machine elements are associative. These are sequential and parallel compositions of operator constructions.

Operations can have not only their own properties, but also properties in relation to other operations. An important property of one operation relative to another is the property of distributivity, which breaks down into two more particular cases. Operationσwill be called distributive on the left with respect to the operationγ, if the following statement is true:

$\sigma(x1, \gamma($ x2 , x3)) $= \gamma(\sigma($ x1 , x2) ,$\sigma(x1, x3))$.

Operation$\sigma$is right distributive with respect to the operationγ, if the statement is true:

$\sigma(\gamma(x2, x3), x1) = \gamma(\sigma($ x2 , x1) ,$\sigma(x3, x1))$.

It is easy to see that if the operation$\sigma$is commutative, and at least one of the properties of left or right distributivity is true for it, then the distributivity property on the other hand will also be true for it. The significance of the distributive property for the study of compositional operations of software machines is determined by the optimization factor introduced by the distributive transformation. This is indeed the case, since distributivity allows us to “bracket out” the general part of the expression, which is some formal recording of a program fragment.

In the properties of operations of universal algebras, it is also customary to highlight properties based on two constants that have certain remarkable properties. These constants are called zero (0) and one (1). In different algebras they may have different notations, but are identical in their properties. There must be an operation for zeroσ, which makes the expression validσ(x, 0) = 0 . For a unit, in the general case, the concept of an inverse element is used. In accordance with it, in the algebra for each element x there must be an inverse element x-1, such that the following statement is true:σ(x, x-1) = 1. A weaker property is the propertyσ(x, 1) = x. This property does not identify an exact unit, but is generally useful for studying the transformation of algebraic statements, which in the case of programmable machines are formal expressions corresponding to programs. The constants zero and one are usually distinguished as 0-place operations to determine the type of algebra.

*Type of algebra*call a set consisting of designations for all operations of this algebra, including 0-place ones, indicating their locality. For example, for formal arithmetic with the operations of addition and multiplication, the following type can be defined:

{ 0 (0), 1 (0), + (2), * (2) }.

Universal algebras having equal types are called of the same type. The corresponding concept can be defined for software machines. Program machines, all the corresponding elements of which are of the same type, are also called homogeneous program machines.

The definitions made allow us to further present the most well-known classes of universal algebras and consider some of their positive properties from the point of view of program formalisms.

Groups

An algebra containing a single binary operation is called a groupoid, and this binary operation is called a multiplication or composition operation. Groupoids do not necessarily require the existence of any operation properties.

In elements of software machines, groupoids can most often be found in algebraic systems of memory elements and interpretation of syntactic constructions, for which a single composition operation is sufficient. Other machine elements are usually more complex and use more than one composition operation.

If the groupoid operation is associative, then such a groupoid is called a semigroup. So, for example, the operation of composition of memory elements cannot be considered associative, since the order of elements following each other is very significant. The operation of composition of elements of interpretation of language constructions is isomorphic to the theoretic-set union, which allows us to consider the universal algebra based on it as a semigroup.

If the signature of a groupoid contains two elements, one of which is a binary operation, and the second is an identity, then such an algebra is called a groupoid with identity. In programming, there are constructs that can be classified as units, for example, the famous empty operator. If we consider its sequential composition with any other operator, while considering the composition to be an operation of

the corresponding groupoid, then the algorithmic algebra based on this operation can be called a groupoid with identity. However, the difficulty lies in the fact that the exact concept of a unit is associated with the existence of an inverse element, which may be true only for some subsets of the tool's software machine. This follows from the fact that almost always in the entire toolkit or algorithmic language one can find a construction for which, based on the result of its execution, it is impossible to determine the initial data, which means it is impossible to find such an opposite operator, a sequential composition with which would give a single operator.

As an example, we can consider the operator of multiplying the contents of two memory elements and writing the result to some third place. Let's denote this operator by f+ (x1, x2, x3) . For a unit to exist in the corresponding algorithmic algebra, the following condition must be met:

f+ (x1, x2, x3)°fx = 1,

Where°- operation of sequential composition of operators. For brevity, the fx operator is written without arguments, since in this case there is no

its location is important, because it is required to find the corresponding operator of any terrain, but with the property f+ (x1, x2, x3) = fx-1 . It is quite obvious that it is impossible to uniquely find such an operator. We can think of the notation f-1 as applying the unary operator -1 to the argument f.

At the same time, there is a subset of operators for which obtaining the inverse operator is not difficult, for example, operators for increasing an argument by a constant value or moving to the next element of a given type. Isolating this subset of operators into a separate subalgebra will allow us to consider it as a groupoid with unity, which provides a significant advantage in optimization and computer-aided design of programs.

A semigroup with identity is called a monoid. It is distinguished from a groupoid with unit by the presence of the associativity property of the binary operation of the signature.

More special cases of semigroups and groupoids are algebras, which have stronger properties of operations. So, for example, a semigroup or groupoid is called idempotent if its binary operation is idempotent and, accordingly, commutative if it has the commutative property. Each of the additional properties provides new opportunities for converting programs, since it puts at the disposal of the researcher a greater variety of possibilities for converting them.

An algebra is called a group if it has type {1(0),-1(1),* (2) } and the following relations hold for its operations:

1) 1 * x = x * 1 for any x from the carrier set,

2) x * x-1 = x-1 * x = 1, for any x from the carrier set,

3) (x * y) * z = x * (y * z) = x * y * z, for any x, y, z belonging to the set of the carrier.

It should be noted that operation designation signs do not play a special role. It is only important to accurately determine their properties.

If a binary operation of a group has the property of commutativity, then such a group is called commutative or Abelian.

For Abelian groups there are many derived properties, proven in various theorems, and allowing complex transformations of the expressions written for them up to the solution of equations. Solving the equations of algorithmic algebras would make it possible to automate many processes of searching for logical errors, optimization and computer-aided design of programs.

Lattices

When considering this class of algebras, it is necessary to consider algebraic systems that include, in addition to carrier sets and signatures, a set of relations on the elements of the carrier set. There are also certain properties for relations that contribute to the analysis of compositional operations research. When considering the elements of program machines, this is very important, since many program composition operations have certain properties only if their elements exist in certain relationships.

For relationships, as for operations, there is its own classification, the basis of which is as follows.

A binary relation r is called reflexive if for any element x belonging to the carrier set of the algebraic system xrx is true.

A relation is called symmetric if for any two elements of the support set x and y the statement xry = yrx is true. Accordingly, a relation is called antisymmetric if the symmetry statement does not hold for it.

If a relation has the property xry & yrz => xrz, where & is logical addition, and x, y, z belong to the carrier set of the algebraic system, then the relation is called transitive. A reflexive and transitive relation is called a preorder relation, and an order relation if it is also antisymmetric.

An attitude is called tolerance or tolerant if it is reflexive and symmetrical. Equivalence is considered to be a reflexive, symmetrical and transitive relation at the same time.

Commutative idempotent groups are called intersection semilattices if they satisfy the following relation:

[(x * y) r (x)] & [(x * y) r (y)],

where * is the binary semilattice operation and r is the order relation. If a symmetric condition is satisfied in which the relation r forms the inverse order, then such algebras are called union semilattices.

An algebra of type { * (2), + (2) } is called a lattice if the following conditions are satisfied:

1) x + x = x for any x belonging to the carrier set,

2) x * y = y * x, x + y = y + x, for any x, y belonging to the carrier set,

3) (x * y) * z = x * (y * z) = x * y * z, (x + y) + z = x + (y + z) = x + y + z for any x, y , z belonging to the carrier set,

4) x * (y + z) = x , x + (x * y) = x, for any x, y belonging to the carrier set.

Under these conditions, expression 4 is called the absorption property.

Lattices are a fairly common type of algebra used in computer programs. In particular, they can be used to describe the properties of operating with set-theoretic data that has order. In some cases, by defining an order that was not explicitly seen in algorithmic algebra, equivalent program transformation

algorithms can be significantly simplified. Lattices can be used to describe hierarchical and relational structures, for example, when analyzing software systems that work with databases.

A well-known type of algebra that is a special case of lattices is Boolean algebras, which are used in the design of logical expressions in program conditionals. These algebras have type { 0 (0), 1(0), -1 (1), * (2), + (2) } and in addition to the conditions that hold for lattices, the following statements must also be true for them:

a) x * x-1 = 0, x + x-1 = 1 for all x from the carrier set,

b) x * (y + z) = (x * y) + (x * z), for any x, y, z belonging to the carrier set.

The operation -1 in Boolean algebra is usually called negation.

Rings

Lattices are complex algebras with several operations. At the same time, real software tools can rarely be classified as lattices due to the large number of properties imposed on the operations of this algebra. Tools can be formally represented by software machines in which algorithmic algebras have complex signatures with various operations having more or less weak properties. In this case, we can consider these operations both separately from each other and their relative properties.

One of the well-known classes of abstract algebras, which can often be used even when studying the compositions of operator constructions in algorithmic languages, is the class of rings.

Let F be some non-empty support set on which two binary operations + and * are given, satisfying the following requirements:

1) < F, + > is an Abelian group,

2) <F, *> - semigroup,

3) for any elements x, y, z of the carrier set there is a distributive property:

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

then the algebra <F, +, *> is called a ring, and its components <F,+> and <F, *> are called the additive group and the multiplicative group of the ring, respectively. A ring can be commutative if the multiplication operation is commutative.

For formal program machines, we can consider more general algebras based on the definition of rings; these include semirings, i.e. rings that do not have a unit, oriented rings, i.e. rings with the only distributive property (left or right), and oriented semirings, which are the result of a combination of these conditions. An example of a program machine whose compositions of syntactic structures correspond to the algebra of an oriented left semiring are programs in extended transition networks implemented in the ATN algorithmic language.

Fields

*Field*is a commutative ring with unity (not equivalent to zero) in which every element has an inverse. The multiplicative group of this algebra is called the multiplicative group of the field.

One of the special cases of fields are fractions with actions on them. Fields can very rarely serve to formalize the control syntactic constructions of software machines, but, at the same time, they can be used in formalisms for the syntactic

constructions of applied operation systems. For example, fields can be used to examine expressions in algorithmic languages that describe various computational constructs.

The considered classes of algebras can be used in various combinations in specific program machines. For some composition operations, local subsets of carrier sets can be identified that meet the conditions of a particular algebra.

When analyzing program machines of algorithmic languages, you can also use the formal means of polybasic algebras. To do this, the carrier set is divided into subsets of different varieties. In this case, any of the signature operations has its own type, which is a list of types to which the arguments of the operation and its result can belong. In this case, each specific argument is assigned its own type of carrier set. Polybasic algebras quite fully describe the syntactic constructions of universal algorithmic languages, since these languages have operators whose arguments can be expressions and other operators of strictly limited types. So, for example, in most languages, the logical expression of a conditional operator cannot contain a cyclic construction, and variable description operators cannot contain switch operators, etc.

At the same time, polybasic algebras are quite complex, but can be reduced to a combination of single-basic algebras, so the question of choosing algebraic formalisms for the study of software machines must be decided by the researcher himself, guided by the criteria of adequacy and simplicity.

Algebras with sorts of supported sets provide an opportunity to evaluate the conceptual integrity of a tool. Conceptual integrity can be understood as fulfilling the requirement of a minimum number of varieties of carrier sets of a software machine while ensuring the completeness of the functional tools of the toolkit for solving the problems for which it is aimed. Thus, the fewer classes of operator constructs of a programming language that must be taken into account in compositions when composing a program, the more holistic the language is.

All tools are objectively subject to continuous development in order to obtain new, more powerful versions. At the same time, when making any changes to them, it is necessary to find out the following properties of the new structures included in the toolkit:

- whether the new design violates the basic syntax of the instrumental system,
- whether new classes of operators appear through additional division of old ones,
- can the newly introduced operator be attributed to any of the already identified classes of operators,
- is there a possibility of violating the structure of programs (given the existence of principles of structure for this language tool),
- whether the new design of the system requires a fundamentally new interpretation.

If at least one of the listed properties of the toolkit is violated, we can talk about a violation of the conceptual integrity of this toolkit by the new design. In this case, by syntax violation we mean the need to use fundamentally new means of syntactic analysis of the language in the corresponding analyzing programs, for

example, compilers. The emergence of new classes of operators means the need for additional division of the constructions available in the toolkit into new classes. For example, you can introduce a new operator into a programming language whose arguments can only be conditional constructs and assignment operators. At the same time, such a separation did not previously exist for other operators, therefore, the new construction will negatively affect the conceptual integrity of the language. In a similar way, a new operator can violate integrity by the fact that it itself belongs to a single new class, i.e. it can only be used in strictly limited arguments to other operators, which was not the case with pre-existing operators. For many tools, the concept of structure makes it possible to build the architecture of the designed program so that it is simply debuggable, changeable and efficient. If a new construction introduced into a toolkit by its developer can, in combination with other operators, turn a program into a non-structural one, then introducing such a construction is inappropriate. The interpretation of the new design also plays a big role. That is, it may turn out that the meaning of its functioning does not fit into the system of concepts that the user of the toolkit subconsciously creates in himself when mentally modeling the work of the program. As an example, we can cite the PROG construction introduced into the Lisp language, which has a non-standard functional interpretation. Its negative impact on the conceptual integrity of the language has been noted even by its developers.

At the same time, objective circumstances may develop in such a way that the introduction of a new design that violates conceptual integrity becomes necessary because it meets certain user requirements. This may be justified if such a construction is not unique, and the system of newly introduced operators has its own conceptual integrity. However, even in this case, the tool developer should develop a completely new tool rather than develop an old version.