FORMAL DESCRIPTION OF A SOFTWARE MACHINE

To obtain a formalism focused on the analysis of the basic structures of an algorithmic language, let us consider the structural components of some abstract language (Fig. 1).

The entire set of tools of the instrumental system forms a software machine (PM), which contains as part of its tools:

- some model of data representation M, mapped to the space of the computer's RAM and external memory,
- constants K, corresponding to the data representation model,
- syntactic constructions of the system of operations A on data structures, used by the program machine to perform the strictly applied part of the program's functioning,
- a set of syntactic constructions F that make up the general syntax of the language constructs of the instrumental system related to controlling the sequence of work of program operators (control syntactic constructions),
- a control system for the interpretation of PM constructions, implementing the basic algorithmic formalism of the language system,
- interpretation of I syntactic constructions of the applied component of the toolkit.

Each of the listed components has its own internal structure and a rigid connection with the other components. The complete set of components M, K, A, Φ, F, I is a formal PM implemented by software in an algorithmic language.

The TL program engine of the L language is as follows
set of algebraic systems:

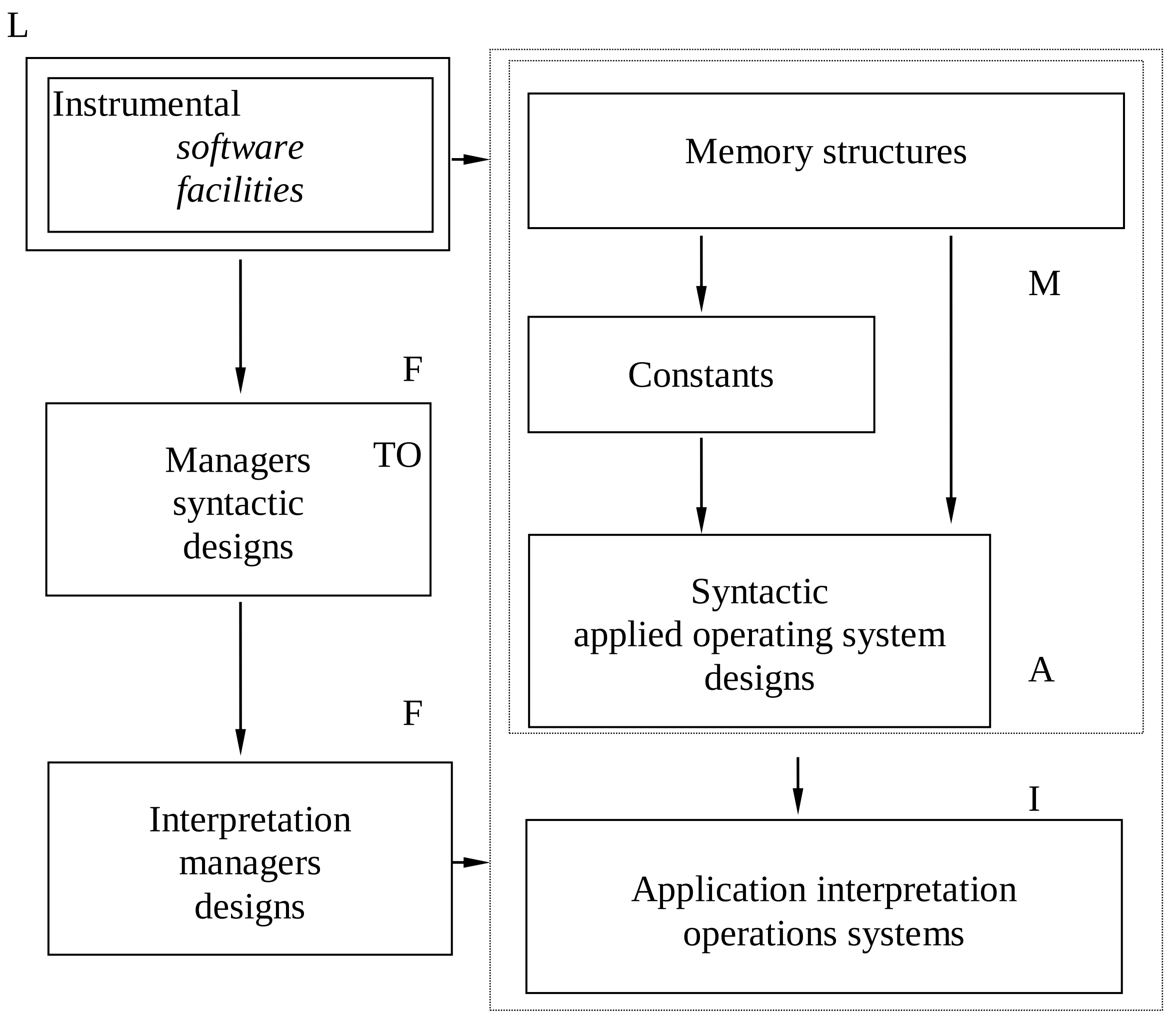
$$T_L = < M_L, K_L, A_L, F_L, \Phi_L , I_L > \qquad .$$

An algebraic system M is an algebra of memory elements with relations defined on the set of its elements.

Memory elements are fragments of RAM used in a particular language for a variety of basic and derived data types, with corresponding memory addresses. So, for example, for languages like C or Pascal, the basic elements of M are addressable memory areas used for objects of integer, real and character types, from which more complex derived structures are synthesized, which also have a single start address of the corresponding memory area.

K is an algebraic system of constants in a programming language. This system is rigidly connected to the M system and corresponds to the applied component of the PM data model of the algorithmic language.

The syntactic constructs of any algorithmic language contain some control constructs for organizing the algorithmic scheme of the program, as well as constructs used to record operations on basic typed objects of the language. To study these constructions in the program machine, the algebraic systems F and A are used, respectively. For universal programming languages, system F includes syntactic constructions such as if-then-else, do-while, case, etc., system A contains operations for numeric and string data types, for example +, *, /, =, etc.



L
Instrumental software facilities
Memory structures
M
Constants
F
Managers syntactic designs
TO
Syntactic applied operating system designs
A
F
I
Interpretation managers designs
Application interpretation operations systems

Fig.1. Structural components of the toolkit

Algebraic systems F and I are intended to describe the interpretation of the corresponding syntactic constructions studied in F and A. The interpretation system of some PL is a description of the semantics (meaning) of its syntactic constructions in some simpler system of commands, operating with concepts that are obviously more primitive than the concepts of this language.

There are two different ways to construct an interpretation system. The first of them is focused on the synthesis of interpretation specialized for a specific algorithmic language. If there is an adequate interpretation system for each language, comparison of various language constructions should be preceded by comparison of interpretation systems oriented to different languages. The second way is to construct an interpretation suitable for language languages of various functional families. The construction of such a system is a complex task, but if it is solved positively, further comparison of language structures is greatly facilitated.

Let us consider in more detail the formal components of PM.

Algebraic system of memory elements

The algebraic system ML of memory elements is the following system of sets:

$$ML = < Am, \Omega_m, Rm >,$$

where a pair of sets $<Am,\Omega_m>$ - algebra of memory elements, in which Am = { m1, m2, ..., mi, ..., mn } - set of addresses of basic and derived memory elements mi, corresponding to certain data types,$\Omega_m= \{\omega_1, ...,\omega_k\}$ - algebra signature, where$\omega_i \in \Omega$are operations on memory elements for the synthesis of derived elements, Rm = {r1, ..., rt } - a set of relations on memory elements reflecting the memory structure of the data model of the algorithmic language L.

The elements of the set Am represent elementary (basic) and complex (derived) memory elements for the corresponding types used in the language. As a result, each of the elements can have its own rank, which characterizes the complexity of the structure of the memory element. A rank is assigned to each of the elements. It represents an integer one greater than the rank of the most complex element included in its structure. Memory elements corresponding to basic data types have a rank of 0. The rank is indicated by the superscript of the element. However, if this does not lead to confusion, the rank index will be omitted from now on.

For most PMs you can use the signature$\Omega_m = \{ + \}$, in which the only operation “+” corresponds to the synthesis of two memory elements into a single derived element of a higher rank.

Using the synthesis operation, it is possible to describe rather complex compositions of memory elements (Fig. 2). In the above figure, the upper index means the rank of a memory element, the lower index means the serial number of the element in the set of elements of a certain rank.

Algebraic relations between memory elements in this case can be expressed by the following equalities:

$$m_0^1 = m_0^0 + m_i^0;\ m_1^1 = m_0^0 + m_n^o;\ \ldots\ m_k^1 = m_i^0 + m_n^0;$$

$$m_0^2 = m_0^1 + m_1^0;\ m_1^2 = m_1^1 + m_i^0;\ \ldots\ m_m^2 = m_2^1 + m_i^1 + m_k^1;$$

or in more detail through basic memory elements, for example:

$$m_i^2 = m_0^0 + m_1^0 + m_i^0 .$$

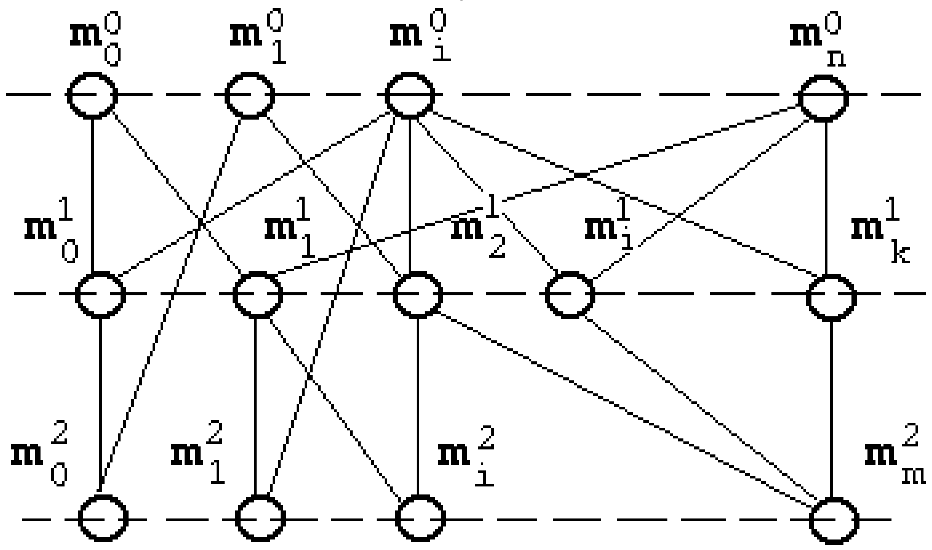


Fig. 2. Graphical representation of algebraic composition

The set of relations over memory elements consists, as a rule, of binary relations and makes it possible to describe various situations of mutual intersection of memory areas, which very significantly affects the properties of the operations of the algebraic system F. We can give examples of the most common relations used in algebraic systems of memory elements for various PLs :

$$Rm = \{ =m, \Omega_m, >m, \geq_m, \leftrightarrow_m, \perp_m \} .$$

In what follows, for convenience of presentation, the lower index of the relation m, characterizing its belonging to the algebraic system of memory elements, will be omitted. For the given set of relations, the first two correspond to the equality and inequality of two memory elements based on the coincidence of their constituent elements. The ">" relation indicates the strict inclusion of all components of the memory element specified by the second member of the relation in the memory element specified by the first member. A complete relationship table includes the following elements:

$ma \geq mb$ - memory element mb is equal to or less than element ma;

$ma \leftrightarrow mb$ - elements ma and mb have common components;

$ma \perp mb$ - elements ma and mb do not have common components;

$ma = mb$ - relation of structural equivalence of types;

$ma \neq mb$ is the relation of non-equivalence of memory elements.

Memory elements can represent not only space for storing any data, but also space for storing program statements. To distinguish such memory elements, the following constructs can be respectively used: m(x) and m(f), where x is some variable or other data-related object and f is some program statement. This notation makes it possible to describe more complex compositions of memory structures, for example, lists, stacks or queues.

Let's give an example of a formal list representation. Let some singly linked list ms contain n elements. Then it can be represented by the following expression:

$$ms = m1 (x1 , m2) + m2 (x2, m3) + ... + (xi, mi+1) + ...+ mn (xn, 0).$$

This expression uses the "," operation, which is a synonym for the composition operation "+". The synonym is used to explicitly highlight different levels of composition in the list, since the operations of including and excluding list elements are operations of a higher level than the connection (xi, mi+1) of the stored element xi with a reference constant to the next element.

Note that the proposed representation of memory elements has one more additional positive quality. It consists in the identical representation of memory elements of different types, but differing only in the way of operating with them. So for example, the representation of queue mq and stack mc would look equivalent:

$$mq = mc = m1 (x1) + m2 (x2) + ... + (xi) + ...+ mn (xn).$$

This representation is fully justified if the stack and queue are implemented not through a list construct, but through an array. The type of the derived data, distinguished by the subscript, is determined by the operations of placing (fct , fqt) and extracting (fcp , fqp) elements:

fct(mx, mc) = mx + mc; fqt(mx, mq) = mx + mq;
fcp (mc = m1 (x1) + m2 (x2) + ... + (xi) + ...+ mn (xn)) = m1;
fqp (mq = m1 (x1) + m2 (x2) + ... + (xi) + ...+ mn (xn)) = mn .

Using the algebra of memory elements, you can define more complex memory transformation operations, for example, the fp operation of including an element in the middle of a list:

fp (ms , xx, i) = m1 (x1 , m2) + m2 (x2, m3) + ... + (xi, mi+1) +
mi+1 (xx, mi+2) + ...+ mn (xn, 0),
where ms = m1 (x1 , m2) + m2 (x2, m3) + ... + (xi, mi+1) + ...+ mn (xn, 0).

Arithmetic operations on memory contents, such as addition of elements, can be defined in a similar way:

f+ (m(x1), m(x2), m(x3)) = m(x1 = x2 + x3).

Algebraic system of constants

The algebraic system of constants is an application system of basic data types used in a language. Most often, this system is polybasic and includes character string algebra, array algebra and formal arithmetic [32,116,117].

In the simplest case, an algebraic system of constants is represented by a system of sets:

$$K_L = < A_K, \Omega_K, R_K >,$$

where <Ak,Ω_k> - applied algebra of constants of basic data types, in which Ak = { k1, k2, ..., ki, ..., kn } - set of constants,Ω_k- many operations $\omega_1, \omega_2, \dots, \omega_j, \dots, \omega_m$, such as, for example, addition, multiplication, concatenation, as well as operations for constructing constants for derived data types.

One of the important types of operations in$\Omega_{To}$are operations for constructing complex constants for derived data types by analogy with the ML system;

Rk = {r1, r2, …, ri, …, rt} - a set of relations of an applied system, for example, formal arithmetic or character string algebra. So, for example, for the algebra of character strings, these can be relations of equality, comparison of strings by length, and inclusion of one string in another.

The algebraic system of constants is rigidly connected with the algebra of memory elements, and for software machines of some tools it is isomorphic to the subalgebra of memory elements. That is, in this case there is a certain subalgebra that is of the same type as the algebra of constants and has equivalent properties of operations from signature sets.

Syntactic constructions of the applied operation system

Algebra AL, used to formally describe within its framework the syntactic constructions of the language tools involved in the descriptions of various more or less complex calculations over variables and constants of basic and derived data types, is based on chains of a certain sublanguage of the language L. These chains are simple expressions, not directly affecting the main algorithm of the program.

As operations for this algebraic system, a system of substitutions is used instead of non-terminal chain symbols - terminal literal constructions.

For example, for arithmetic expressions, the operations of addition, subtraction, multiplication and division can be defined with the following syntax, similar to the syntax of the Lisp language [13,118]:
(ADD List1 List2), (SUB List1 List2),
(MUL List1 List2), (DIV List1 List2) .
For these operations, corresponding program terms can be defined:
fadd (x1, x2), fsub (x1, x2), fmul (x1, x2), fdiv (x1, x2).
Three-argument composition operationσcan be represented by the following expression:
fx: fx (x1, ..., xi, ..., xn), fy: fy (y1, ..., yj, ..., ym),
σ(fy, i, fx) = fx (x1, ..., fy, ..., xn) .
We will further call such a composition substitution. For syntactic constructions of software tools, it is important and allows the formation of complex syntactic expressions, for example:
(SETQ X (ADD (MUL YZ) X),
which in the form of a program term looks like this:
fsetq (x, fadd (fmul (y, z), x)) .
It is easy to see that in this expression the substitution composition operation is used twice:
σ(σ(fmul (y, z), 1, fadd (xi, x)), 2, fsetq (x, xj)) =
σ(fadd (fmul (y, z), x), 2, fsetq (x, xj)) = fsetq (x, fadd (fmul (y, z), x)) .

Control syntactic structures

Control constructs of an algorithmic language serve to branch the linear computational process in order to execute the most complex algorithmic part of the programs for which this language is oriented. The corresponding algebraic system describes the possible compositions of syntactic constructions of PL or other software tools used to specify the structure of the algorithm. It represents the following system of sets:

$$FL = \langle Am, \Omega_m, Rm \rangle,$$

where a pair of sets < AF,Ω_F> is an algebra of control structures, in which AF = { f1, …, fi, .., fn } is a set of command structures described by the generative grammar GL of some language L and characterizing the control of calculations in the program. Each construction fi can have an arbitrary locality (fi) = n, indicating the possibility of performing n substitutions in those places fi that correspond to non-terminal symbols of the GL grammar.

The substitutions themselves are elements of many operationsΩ_F, necessary for the formation of more complex and more specific programs. So, for example, the design

if x0 then x1 else x2;

can be represented as

f1 (x0, x2, x3), (f1) = 3,

and the construction while x0 do x1 ; How

f2 (x0, x1), (f2) = 2,

compound operator

Begin x1, ..., xn End;
as an n-ary construction

f3 (x1, ..., xn) .

The RF set consists of binary relations over the control constructions $=,\neq,\geq,\leq$, $>$, $<$, having the following meaning:

= - equivalence of structures,

≠- non-equivalence of structures,

≥- non-strict inclusion of one construction into another,

≤- the opposite relationship to the previous one,

>, < are strict occurrence relations.

Depending on the subject of the study, the listed relations may have semantics of varying precision. For example, one can consider the equivalence of constructions up to their complete syntactic coincidence, up to the functional equivalence of their actual interpretation in programs, or up to the preservation of a predetermined property by the program.

Interpretation of syntactic constructions

Interpretation of syntactic constructions is a way of specifying the semantics of constructions through the command system of some simpler PM. Such a machine can be, for example, a processor of some computer (special processor) or the basic assembler of a computer.

The advisability of a formal consideration of the interpretation of syntactic constructions is determined by the following reasons:

1) it becomes possible to prove the correctness of the composition of various syntactic structures of the language into more complex structures;

2) it becomes possible to prove the redundancy or sufficiency of a set of syntactic constructions of an algorithmic language, as well as the substantive integrity of its syntactic system;

3) strict consideration of the interpretation of constructions leads to optimization of the program compilation algorithm by considering injective mappings of syntactic constructions into a simpler system of operations, as well as equivalent optimizing transformations at the level of the interpreting system (program machine);

4) the equivalence of some complex syntactic structures can be proven through interpretive sequences [90,91,92].

Just as for syntactic constructions, for the algebraic interpretation system a division is introduced into the interpretation of control syntactic constructions and the interpretation of the applied system of operations.

Algebraic systems that describe interpretation have carriers with elements - n-ary terms, in which the following types of constructions are used in place of arguments:

m(xi) - address of the first element of the object (variable) with the name xi,

m(Ii) - address of the operation named Ii,

m(Stop) - address of the empty operation that completes execution,

m(φi) - transaction addressφ_i.

The "Stop" symbol will mean an empty final operation of interpretation of any syntactic construction and can be omitted (absorbed) during the sequential composition of two interpretive sequences of operations.

Application Operation System Interpretation

The applied system of PL operations involves the calculation of expressions that are allowed in the area of defining the basic and derived data types of this language and do not directly lead to branching of the algorithmic structure of the program. This fact indicates the relative independence of the algorithmic structure of the program from expressions written within the framework of the applied system of operations, which allows us to consider these two components separately, exploring the relationships of the corresponding monobasic algebraic systems [119,120].

The interpretation algebra of the applied IL operations system is the following system of sets:

$$IL = < AI, \Omega_I, RI >,$$

where AI is a set of microprograms that interpret expressions of the application operating system. For this set there is a certain subset $AI0 \subset AI$, called the basis of the interpretive system, those. a set of elementary operations from the composition of which all other microprograms are obtained.

For example, for formal arithmetic used in universal algorithmic languages, these can be operations “+”, “-” with stack memory operations and unconditional jumps attached to them:

$$AI = \{ I+, I-, I v \} ,$$

where I+(m(xi), m(Ijx)) is the addition of xi with the contents of the adder and placing the result in the adder, then proceed to the operation Ijx; I- (m(xi), m(Ijx)) - operation of decreasing the contents of the adder by the value xi, Iv(m(xi), m(Ijx)) - transferring the contents of the adder to the variable xi.

A bunch ofΩconsists of a single sequential composition operation “◦“ interpreting expressions into derived expressions. For example, resetting the adder can be described as follows:

$$[Iv\ (m(x), m(I-))]^\circ[I-\ (m(x), m(Stop)] = I^{\Sigma}(m(x),m(Stop)),$$

where m(x) is the memory allocated for an arbitrary temporary variable.

Sequences of compositions allow you to describe the interpretation of arithmetic expressions in PL. As an example, we will describe the arithmetic expression in C language x = a + b; .

This expression uses two C operations of different priority: { =, + } or { f=, f+ }. In the form of terms of the algebra of constructions of the syntax of an applied system of operations, the expression in question can be written in the form f= (x, f+ (a, b)).

The interpretation of this expression is as follows:

$$I^{=+}(x,a,b) = I^{\Sigma}(m(x),\ m(Stop))\ o I^{+}(m(a), Stop)\ o$$
$$o I^{+}(m(b), Stop)\ o I^{v}(m(x), Stop).$$

Interpretation of control syntactic structures

Control syntactic structures described by the FL system have the following FL interpretation system:

$$FL = < AF, \Omega_F, RF > .$$

Here AF is a set of interpreting elements (microprograms used to interpret the control structures of the tool L),Ω_F- a signature consisting of composition operations (usually a single operation of sequential composition "°"), RF - relations of equivalence and occurrence on interpretive sequences from AF.

As in the IL system, for AF there is a basis subset $A0 \subset AF$, which consists of compositionally indecomposable elements.

This basis is the basis both for comparing the functional concepts of different language languages, and for studying the feasibility of various language constructs within the framework of one algorithmic language. The most powerful bases are the famous Post and Turing machines.

At the same time, to study the properties of constructions in modern languages, it makes sense to consider PMs of lower power with more complex bases.

Let us propose one of the bases, which we will call A0T

$A0T = \{\varphi_0, \varphi_1, \varphi_2, \varphi_n, \varphi_d\}$, $(\varphi_0) = (\varphi_1) = (\varphi_2) = (\varphi_n) = (\varphi_d) = 2$.

The functional meaning of the elements of the set A0T is as follows:

$\varphi_0(m(f),\varphi)$ - execute a sequence of commands located at address m(f), then proceed to execute the commandφ;

$\varphi_1(m(\varphi_0),\varphi)$ - execute a terminal (application) command located at address $m(\varphi_0)$, then proceed to execute the commandφ;

$\varphi_2(m(\varphi_0), m(\varphi_1))$ - if the result sign is greater than zero, proceed to executing the commands located at address $m(\varphi_0)$, if less than or equal to zero - to address $m(\varphi_1)$;

$\varphi_n(m, m(\varphi))$ - allocate memory m, proceed to executionφ;

$\varphi_d(m, m(\varphi))$ - free memory m, go to executionφ.

Note that the proposed basis allows you to write down the interpretation of various program constructs in any order, since each of the commands included in it has a link to the next command as one of its arguments. Moreover, in the case of interpreting real program designs, double indexing must be used to identify each command. For example, the commandφ_d^1describes the first occurrence of the command$\varphi_d$into an expression of interpretation of some program fragment.