DSA-THEORY. PRESENTATION OF CANONICAL ALGORITHM BY MEANS OF ALGORITHMIC LANGUAGE

V.G. Kolesnyk

This work continues the description of the decomposition scheme as a theoretical model, which makes possible generation of the applied algorithms. The description of the algorithmic language made for show the possibility of algorithms generation is given. One of the factors' group is described, i.e. ways of placing the properties on the tape of the abstract type which when is taken into consideration allows to turn canonical algorithm into a real one and applied algorithm, which is the same as turning the decomposition scheme into the program text. The notions of algorithmic primitive and algorithmic joint (operand and operation) as the means for constructing the algorithm are introduced. These notions of algorithms construction are the alternatives for the notional system and methods of structured programming. The notions of functional core, algorithmic frame, functional contents and algorithmic matrix are introduced.

Introduction

In [1], a description of a theoretical model called a decomposition scheme is given. It is also shown there that the decomposition scheme (DS) is of an algorithmic nature. In [1], the synthesis path (PS) and the canonical (universal) algorithm (CA) are described as the main attributes of the decomposition scheme.

DSA-theory [1], of which DS is an integral part, as a theoretical model, creates the prerequisites for generating algorithms (not machine code). Despite the fact that PS and CA are quite formalized, in order for them to be implemented in practical processing as a computer program, they must be represented by means of an algorithmic lowercase language. This article introduces a description of the algorithmic language. The language is not intended for practical programming, but only to demonstrate the possibility of generating algorithms, to demonstrate the text of the generated program.

The paper introduces the concept of an algorithmic primitive, which is used as a component in the construction of a DS. The properties of algorithmic primitives are described. The constructions of operators and sentences in an algorithmic language are given, with the help of which algorithmic primitives are represented. The concepts of algorithmic joints as operations on algorithmic objects are described.

A CA that can be built on the basis of DS, on the way to its transformation into a real program, must be modified taking into account the specific computing environment in which the program will operate. Those factors (algorithmically relevant factors) that need to be taken into account are diverse and there are quite a lot of them. From the point of view of DSA-theory, they are grouped. The article describes the procedure for making changes to the CA generated by one group of factors – the methods of placing P-data on the A-tape.

Despite the fact that the algorithm as a result of generation is represented using an algorithmic language of an imperative type, DS, as a source description for generation, is a description of a declarative type. The generated algorithm, by definition, reflects the order of computations, and the description of DS does not explicitly show this order. In the process of generating the algorithm, the order of traversal of the tree of the complete decomposition scheme (DPS) from top to bottom and from left to right is taken into account, and due to this it appears in the generated algorithm. In fact, DS is an object of declarative programming, although it has no outward resemblance to traditional declarative programming languages [2-3].

DS Algorithmic Components

Next, the concept of a decomposition scheme primitive (SDP) is used. The simplest elements, or primitives, from which the decomposition scheme is built are the analytical algorithmic dependence (AAD), synthetic dependencies (SAD): the first (SAD^1) and the second type (SAD^2). Decomposition mechanism (DM) is also considered as SDP. More complex structures are built from

SDP. The prototypes of SDP are the basic structures of structured programming [4].

The SAD consists of two parts: a header and a body. The heading defines the start and end conditions for the work of the SAD, and the body performs work with the next component of the C-property [1]. Let two nodes be connected by a branch (family relationship). A certain SAD is associated with them. If a SAD is associated with a parent node, then the SAD header will be executed in the list of A-dependencies associated with this node, and the body of the SAD will be executed in the list of A-dependencies associated with the child node.

The decomposition mechanism is represented by a set of sentences DM = (S_{begin}, S_f, S_{next}, S_{end}), where S_{begin} is a condition for starting decomposition, S_f is a description of the procedure for the first decomposition step, is a description of working with the first part, S_{next} is a description of the procedure for the next decomposition step, S_{end} is a description decomposition termination conditions. S_{begin}, S_f, S_{end} - these three components are called the header of the decomposition mechanism. S_{next} - this component is called the body of the decomposition mechanism.

AAD, heading SAD and DM, body SAD and DM are called the simplest settlement procedures (SSP).

An different number of A-dependencies can be associated with the leaves and nodes of the DPS. If the result of the implementation of one A-dependency is the initial data for the implementation of another A-dependence, then the A-dependencies are interdependent. When writing such A-dependencies, both the order of their enumeration and the order of their execution should be preserved. If the order of implementation of A-dependencies is absent, then A-dependencies are independent of each other.

It is possible that a settlement operation that implements the A-dependence can be performed only under a certain condition. This is due to the fact that there is a conditional type of a part of the object; at the time of decomposition, some P-property may be missing. Because of this, some A-dependence will not be calculated (implemented) in the PS at the time of calculation. Such A-dependencies are called conditional.

Two conditional A-dependencies can be related. The essence of their relationship lies in the fact that the calculation procedures that implement them are performed depending on certain values of some P-property. There can be more than two such conditional A-dependencies. They are called alternatives.

An arbitrary number of AADs, DM headers, own SAD headers can be associated with a DPS node that is not a leaf node. An arbitrary number of AADs, a SAD body, and one DM body can be associated with a node that is not a root.

Considering that an arbitrary (large) number of AAD, SAD components, and DM components can be associated with each of the leaves, it is necessary to preprocess the entire set of SSPs. To do this, do the following.

1. Dependent SSPs are collected and combined in the order of their mutual dependence. After that, this algorithmic construction (AC) is designated as a single whole.

2. Conditional SSPs to collect and link. The principle of union is that they have the same or different value of the same El-property. As a result, AC is also formed, which should be designated as a single whole.

3. Alternative SSPs to collect and merge. As a result, AC is also formed, which should be designated as a single whole.

AC and SSP, as well as A-dependencies, can also be conditional and unconditional, and can also be dependent or independent.

The previous two steps are repeated several times. But both SSP and AC are subject to analysis and integration as components. Of these, more complex ACs can be formed. The analysis and merging will be completed when the list of calculation procedures lists all DPS and SSP associated with one node, and complex ACs that are independent of each other. The list of these independent PRP and AC is called the algorithmic construction of the node (leaf) - (ACN). At DPS, each node and leaf will have its own ACN.

A more precise definition of the CS is as follows: the synthesis contour is a list of ACN,

compiled in the process of bypassing the DPS from top to bottom and from left to right. Compilation of the PS as a list of ACN is the first step in the synthesis of the final algorithm, which is suitable for processing in a computer.

P-properties and P-data

In [1] the definitions of El-property, A-property, C-property and their generic concept - P-property are proposed.

A C-property consisting of one El-property is called simple. A-property, which consists of an arbitrary (more than one) number of El-properties, is called simple. A C-property that consists of a simple A-property is called an extended property. An A-property that contains one or more simple or extended C-properties is called an extended property.

A C-property that consists of an extended A-property is called complex. Only one A-property is associated with each type of part of an object or the object itself. This A-property is called the main A-property for this type of object part. The main A-property can consist of intermediate A-properties that are included in the main as components. Intermediate A-properties can contain A-properties that are also intermediate. That is, there is a hierarchical structure of A-properties.

P-properties can be fixed on media available for machine processing. In this regard, along with the P-properties of the object, P-data about the object are also considered. Data about an object (P-data) or a given quantity characterizing an object is a property of an object for which a sign is defined that can be read or perceived by a person and/or technical means. Similar to the concepts C-property, El-property and A-property, there are concepts: aggregate data (C-data), aggregated data (A-data) and elementary data (El-data) and their generic concept – P-data. P-properties and P-data refer to both the object and the parts of the object. A-data may include A-data and C-data along with El-data. C-data may contain A-data. Extended C-data and A-data, complex C-data and A-data, and the main A-data are also considered.

Each part of the object must be named. El-data, which contains the identifier of an object or part of an object, must be present in the main A-data. As part of the A-data, this identifier is called the key. A key is a generalized concept of a record key traditionally understood in file and database processing. Specific numeric, textual, and other values of P-properties and P-data are called D-data. The prototype of the D-data concept in electronic data processing is simply the given.

D-data carrier

Two types of D-data carriers are considered: A-memory and A-tape. A-memory is a generalized abstract concept of computer RAM. Access to D-data located in A-memory is realized by mentioning the name of the P-data. The concept of "A-memory" is simplified to the limit and is intended for modeling algorithms that implement A-dependencies.

A-tape is an abstract image of a tape (magnetic or paper), consisting of cells, along which the reader (A-head) moves. In this paper, the notion of an A-tape becomes more complicated than in [1]. It is assumed that the cells of the A-tape are grouped into records. The size of a record is determined by the size of the simple A-data contained in the record. A-tape can be divided into sections that have more entries. That is, there is a hierarchical structure of nested sections of the A-tape. These areas are called regions and sub-regions.

Moving the A-head forward performs reading, while moving backwards only rewinds. The A-head can read records from A-tape and transfer to A-memory, or select from A-memory and write records to A-tape. After the record has been read into the A-memory, access to the D-data becomes possible. Access to records on the A-tape is only sequential. That is, in order to read the tenth record, you must first read nine records preceding the tenth. Using the A-tape, the storage of D-data is modeled and the process of generating algorithms is illustrated.

The types of records, areas and sub-areas on the A-tape must be distinguishable. The types of regions and sub-regions of the A-tape are named. For example: capital Latin letters with subscripts (or without).

The fact that a region or sub-region of some type contains an arbitrary number of records or sub-regions will be marked with a μ at the superscript level next to the name of the region type. Or R^{μ}=<R1>, which means “Region of type R (or region R) contains an arbitrary number of sub-regions of type R1”.

The fact that a region or sub-region contains a certain number of records or subregions will be marked with a ν at the superscript level next to the name of the region. R^{ν}=<R2,R3,R4>, which means “region R contains three subregions R2, R3 and R4. Each of them meets once”. The order in which the subregions are listed in this sentence corresponds to the order in which they are placed in the region. If this is not the case, then a different procedure is stipulated.

To describe how the regions and subregions of the A-tape are arranged and nested, sentences of the two types described are used. All sentences together make up the tuple RG, which describes the structure of regions and subregions on the A-tape. The placement of regions and subregions can be represented as a tree.

Placement of P-data on A-tape

There are several types of placement of P-data in regions and sub-regions.

A simple A-given. The smallest indivisible part of the A-tape is the record. A simple A-data is placed in the record. The fact that an A-data Q is placed in a notation R is written as R[Q].

Records and subregions divide the A-tape into fragments. Hereinafter, these fragments are called A-fragments.

Extended C-data. An extended C-data occupies an area or subarea consisting of records. One record contains one A-data, which is a component of the C-data. The region or subregion occupied by the extended C-data is called simple.

Example: An object has an aggregate property RA, which consists of an El-property AD (fig. 1). Area D contains an arbitrary number of R records. The elementary property AD is placed in the R record. Thus, the aggregate property RA is placed in area D.

FS: RA = {∪AD}.

RG: D [RA], R [AD], D^{μ} = <R>.

Since the P-properties RA and AD have a location defined, they are also P-data. Although after placing P-properties, their names can not be assigned to the corresponding P-data, but they can be assigned other names.

A-fragments must be visible on the A-tape. For A-fragments, start and end attributes must be defined.

A sign of the beginning of a subarea can be:

- Service record - a marker.
- Some service El-data (record type), which is contained in each C-data record. This El-data has a certain meaning. If the records of at least one C-data contain an auxiliary El-data, then it must be in all records (components) of the original extended A-data.
- Changing the value of a key. That is, in all records (components) of the original A-data there is an El-data with the same value. It is characterized by the fact that for one C-data, this El-data has the same value.

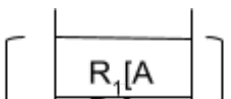
The sign of the end of the subarea can be:

- Service record - a marker.
- Changing the value of service El-data (record type).
- Changing the value of a key.
- End of A-tape

Ways to place A-fragments

Ways to place A-pieces inside the A-tape area can be as follows:

Order relative to placement area boundaries (ORb). All A-fragments containing



extended A-data components are assigned a sequence number relative to the beginning and/or end of the area (Fig. 2).

Example:

FS: Z = {A,B,C,D,E}. A, B, D are simple A-data. C and E are simple C-data.

RG: R^{v}=<R1[A],R2[B],R3[C],R4[D], R5[E]>, $R3^{\mu}$=<R31>, $R5^{\mu}$=<R51>. Here R1, R2, R4, R31, R51 are records, R3, R5 are subareas.

R1 and R2 may or may not have an entry type.

All other records must have a required entry type. If there is no record type, then there must be an El-data that acts as a key, and for records R4, R31 and R51 key values must be defined.

For all A-fragments, sequence numbers are set in the A-tape zone: R1 is the first, R2 is the second, R3 is the third, R4 is the fourth, R5 is the fifth. The beginning of subarea R3 is the first record R31, the end of subarea R3 is the record R4. The beginning of subarea R5 is the first record of R51, the end of subarea R5 is the end of the A-tape.

Relative order (ORn). Two or more A-fragments have an order relative to each other.

Example:

FS: Z = {A,B,C,D,E}. A, B, D are simple A-data. C and E are simple C-data.

RG: Rv=<R1[A],R2[B],R3[C],R4[D], R5[E]>, $R3^{\mu}$=<R31>, $R5^{\mu}$=<R51>.

Here R1, R2, R4, R31, R51 are records, R3, R5 are subareas.

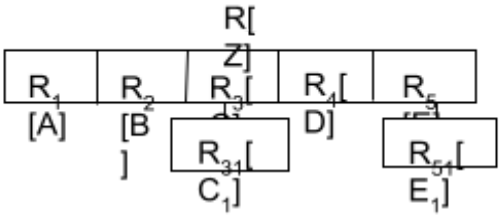


Fig. 2. Extended A-data components have sequence numbers relative to the beginning of the region.

Subarea R3 follows immediately after R2. R31 and R51 records are compact and must have an entry type or a key with a specific value. The beginning of the R5 subarea is the first record of R51, the end of the R5 subarea is any record different from R51. If the subarea R5 is at the end of the A-tape, then the sign of the end of the subarea will be the end of the A-tape. The beginning of the subarea R3 is the record that immediately follows the record R2. The end of the R3 subarea is any record other than R31. If the subarea R3 is at the end of the A-tape, then the sign of the end of the subarea will be the end of the A-tape.

A placement method is possible when two A-fragments do not directly follow each other. Such a situation may arise when A-fragments are not involved in processing and must be skipped when reading A-data from an A-tape. This is the so-called unproductive D-data.

Order by key value (ORs). A simple subarea containing A-fragments can have a characteristic – the order of its components. Records can be sorted by the values of one or more E-data included in the A-data contained in the record. Records can be sorted in ascending or descending order by key. There may be more complex relationships between record keys. The order between records is a characteristic of the subarea.

The placement of extended A-data can be varied. The reasons for this are as follows:

• Extended A-data can consist of simple A-data. The latter are placed in records and interspersed with extended C-data. The so-called intermediate A-data.

• The number of intermediate A-data can be arbitrary.

• The number of intermediate C-data can be arbitrary.

Combination of different ways of placing A-fragments

The components of the extended A-data, when placed on the A-tape, may have a different order. There are several ways to co-locate components.

Collocation of the first kind (mix1). For A-fragments, the placement order is not established.

FS: Z = {A,B,C,D,E}. A, B, D are simple A-data. C and E are simple C-data.

RG: R^{v} = <R1[A], R2[B], R3[C], R4[D], R5[E]>, $R3^{\mu}$=< R31>, $R5^{\mu}$ = <R51>, R1, R2, R4, R31 , R51 – records; R3, R5 are subareas.

The situation is shown in Fig. 3. All records must have a record type. Each of the subareas R3 and R5 is placed compactly. The beginning and end of each of the subareas must be defined.

The beginning of the region R is one of the records R1, R2, R4 - or the beginning of the subareas R3 (record R31), R5 (record R51). The end of R is the end of an A-tape or a record other than R1, R2, R4, R31, R51.

The beginning of R3 is a record R31, the end of R3 is a record that differs from R31. The beginning of R5 is a record R51, the end of R5 is a record that differs from R51.

It is written like this:

RG: R=<R1[A]° R2[B]° R3[C]° R4[D]° R5[E]°>; $R3^{\mu}$ =<R31>; $R5^{\mu}$ = <R51>. Here R1, R2, R4, R31, R51 are records; R3, R5 are subareas. The ° sign next to the names of records or subareas indicates that these records or subareas are jumbled.

Joint placement of the second type (mix2). Some A-fragments may have a relative order, while the rest have an arbitrary order. In this case, placement is performed in the so-called work sub-areas. The grouping of components is performed. If any components have an order relative to each other and are placed side by side, then they are combined into a subarea. As a result, the work sub-areas and the rest of the components are co-located mix1.

FS: Z = {A,B,C,D,E}. A, B, D are simple A-data. C and E are simple C-data.

RG: R^{v}=<R1[A]° R6° R4[D]° R5[E]°>, $R3^{\mu}$ = < R31>, $R5^{\mu}$ = <R51>, here R1, R2, R4, R31, R51 are records; R3, R5 are subareas. R6 = <R2[B], R3[C]> is work subarea.

Co-location of the third type (mix3). At the same time, some A-fragments have a certain place relative to the boundaries of the placement area, while others can be mixed. R1 is the first, R2 is the last, and the record R4 and the subregions of R3 and R5 are mixed in one area. The records R31, R4 and R51 must have a type. The sign of the beginning in the area where they are stored is the first one encountered among R31, R4 and R51. The subarea terminator is the end of an A-tape or a record other than R31, R4, and R51. If some subareas or records have an order relative to the scope boundaries, then they can be at the beginning or end of the containing area.

FS: Z = {A,B,C,D,E}. A, B, D are simple A-data. C and E are simple C-data.

RG: R^{v}=<R1[A], R3[C]° R4[D]° R5[E]° R2[B] >, $R^{\mu}3$=< R31>, $R^{\mu}5$ =<R51>, here R1, R2, R4, R31, R51 - records; R3, R5 are subareas.

Joint placement of the fourth type (mix4). A-fragments can have a relative order, an order relative to the boundaries of the region, or (and) an arbitrary order.

FS: Z = {A,B,C,D,E,F,G}. A, B, D, F, G - simple A-data. C and E are simple C-data.

RG: R^{v}=<R1[A], R2[B], R3[C], R4[D], R5[E], R6[F], R7[G] >, $R3^{\mu}$ = < R31>, $R5^{\mu}$ = <

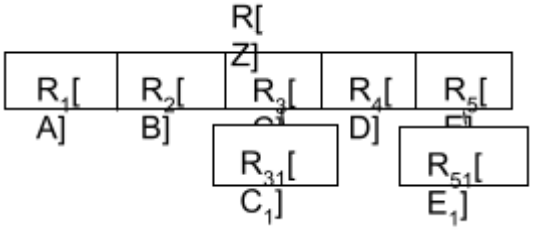


Fig.3. Extended A-data components are co-located

R51>. R1, R2, R4, R31, R51, R6, R7 - records; R3, R5 are subareas.

R1 is the first record in the area - R, R6 is the penultimate record, and R7 is the last. R4 and R5 - follow directly one after another.

In this case, nesting is performed in work subareas. A grouping of A-fragments is performed. If any A-fragments have an order relative to each other and are placed side by side, then they are combined into a subarea. In the case of a specific example, this type of placement is written as follows:

RG: R^{v}=<R1[A], R2[B]° R3[C]° R8°, R6[F], R7[G] >, $R3^{\mu}$ = <R31>, $R5^{\mu}$ = <R51>, here R1, R2 , R4, R31, R51 , R6, R7 – records; R3, R5 are subareas. R8 = < R4[D], R5[E] > – working subarea.

Input and output A-tapes

The existence of A-dependencies and PS suggests that there are P-properties (respectively, P-data) that are initial for calculations and P-properties that are the result in calculations. If AAD is implemented, both operands and results are placed in A-memory and then output in the same record. But it is possible that before the calculation, the P-data are placed on one A-tape, and the result of the calculation should be displayed on another A-tape. The DSA-theory provides for the possibility of working with P-data placed in the input and output A-tapes. The A-tape used to store the source data for calculating the P-data is called the input. The A-tape used to store P-data, which is the result of calculations, is called the output tape.

Representation of SDP by means of an algorithmic language

Due to the fact that the placement of A-fragments containing P-data has various variations described above, the CA is a more complex design compared to that described in [1]. Further, the CA will be described using an algorithmic language similar to the early versions of the COBOL language. To describe the algorithms, an algorithmic language is used, syntactic constructions, which are introduced as needed. The language does not have a strictly formal definition, but is based on an intuitive understanding of traditional algorithmic languages. In the course of the presentation, the definitions can be refined to the minimum required level. The language assignment is temporary.

Implementation of AAD. The smallest and main part of the PS is the action that implements one AAD. The action that AAD implements is described by an assignment operator. More precisely, the assignment operator that exists in traditional algorithmic languages in their simplest form.

El-properties A and B connected by an equal sign A = α (B) are implemented as an assignment operator A:=B, which means: “Based on El-property B, in accordance with the dependence α existing between A and B, an El-property A is being formed". Dependence α may contain traditionally used signs of arithmetic and algebraic operations, separators. If, in the course of the presentation, there will be a need to use algebraic functions that are implemented by subroutines in traditional algorithmic languages, then this will be specifically discussed. Also, the situation will be specifically stipulated when AAD exists, but cannot be expressed by a composition of arithmetic and (and) algebraic signs. A special case of the assignment operator is the transfer of a value from B to A.

The statements generated by AAD can also be dependent or independent.

Algorithmic joints. The following describes ways to combine algorithmic language statements into more complex constructs. The union is performed using the so-called algorithmic joints - sequential, conditional and alternative. Articulations are operations on algorithmic objects (and, accordingly, on operators and sentences in an algorithmic language).

A group of statements is called a sentence. The sentence ends with a period or is delimited by operator brackets, which will be described below. The proposal can be divided into parts only from the point of view of convenience of perception. Within a sentence, operators are separated by a comma or space(s).

Operators in a sentence are joined by **sequential** connection. The sentence is further considered as an indivisible construction, and can be used as a component in further connection, resulting in complex sentences. The sentence fixes the order of execution of operators and, accordingly, AAD, regardless of what they were before linking - dependent or independent.

Conditional AAD or SSP are represented by conditional statements - (IF THEN). Conditions are described using traditional comparison signs (>, <, =, ≥, ≤, ¬), logical operations (OR, AND, NOT) and delimiters (space, paired brackets). The names of P-data are used as operands in comparison operations.

Example: IF L=N THEN A := B ENDIF or IF L=N (A := B).

Here, the "THEN" and "ENDIF" operator brackets and parentheses pairs perform the same function. There can be more than one statement inside parentheses. In the case of the previous example, the operator is called conditional.

If two or more PRPs are satisfied when the same condition is true, then the corresponding statements are combined into a clause. Such a sentence is conditional, and the connection of statements is called conditional connection.

If the terms of two or more conditional sentences arranged in series (not necessarily) mutually exclude each other, then they are also combined into one sentence. Such a sentence is called an alternative sentence, and the connection is called an **alternative** connection. The prototype of the alternative connection is the CASE operator in traditional algorithmic languages.

Example: Three conditional sentences

IF A=1 (B := C).

IF A=2 (B := 2*C).

IF A=3 (B := 0).

Articulated into one alternative.

```
CASE
        IF A=1 (B := C)
        IF A=2 (B := 2*C)
        IF A=3 (B := 0)
ENDCASE
```

Here CASE and ENDCASE are operator brackets that delimit a conditional alternative sentence.

If only operators are articulated in conditional or alternative sentences, then these sentences are called simple conditional. Simple conditional sentences can be considered as indivisible wholes and as components (operators) can be combined into more complex structures.

A sentence made up of joints of at least two of the above types and having a sentence as a component, and not just operators, is called a complex sentence. A complex sentence can also be considered as an indivisible whole and used as a component. Thus, a complex construction, composed of operators, having a hierarchical structure, can be generated.

Conditional, alternative and complex sentences can also be dependent or independent.

DM implementation. To write DM procedures, several types of algorithmic constructs are used, in which the read operator (READ) and the write operator (WRITE) are used. The prototype of these operators are input-output operators in traditional algorithmic languages.

The read operator, READ, uses a pointer that indicates whether or not the A-tape has ended. If, when trying to read from the A-tape, the next record is read and the corresponding A-data is available in the A-memory, then the value of the pointer is “Y”, otherwise “N”. The input statement works like this:

If another record is available while trying to read from the A-tape, then the A-data contained there is transferred to the A-memory. The A-head moves one record forward. The pointer is set to the “Y” state. If there is no next record on the A-tape, the A-head does not move, and the pointer is set to the “N” state. The name of the pointer is D_An, where n is the ID of the A-tape.

The repeated execution of an input statement is implemented using a construct that is called a loop in traditional algorithmic languages.

In most algorithmic languages, the loop head and body are textually inseparable. The temporary language represents the loop operator with two sentences, the connection between which is established by the identifier. This is a difficult proposition. This is similar to the relationship between the header and the loop body in early versions of COBOL.

PERFORM AA (condition)

.

.

AA: BEGIN… END AA

Where PERFORM is the name of the loop statement. AA is the name of the cycle body, by which the connection between the heading and the body of the cycle is realized. The condition in brackets is the condition upon reaching which the execution of the loop stops. The number of repetitions can also be specified here. BEGIN and END are operator brackets that delimit the loop body.

Below is an example of placing a specific C-data in an area (Fig. 4).

FS: Z = {B}, B is a simple C-data.

B = {∪ C}, C is a simple El-data.

RG: R^{v} = < R2 [B]>, $R2^{\mu}$ =< R21[C] >.

A-tape has ID A1. DM is represented by operators as follows.

PERFORM AA ((D_A_{A1} = "N"))

.

.

AA: BEGIN

READ A1

ENDAA

When implementing procedures, the DM may have a start condition - Sbegin. In the text of the algorithm, the condition is realized by a conditional represented by an operator or a sentence. When implementing a DM, there may also be initialization and (or) termination procedures.

After the simple D-data has been moved from the record to the A-memory, the components of the A-data are accessed by name. The input statement specifies the name of the A-tape identifier.

Implementation of SAD. SAD1 is implemented using a cycle. Part of this cycle, more precisely, part of the body, is an action that saves each implementation of El-data C on the A-tape.

Example: B is a simple C-data; C - simple El-data; B = {∪C }; E – El-data placed in A-memory;

RG: R[B]; R^{μ} = < R_1[C] >;

R is placed on the A-tape, whose ID is A1. There is a relationship between D and B. The algorithmic construction consists of two parts.

PERFORM BB ("completion of calculations").

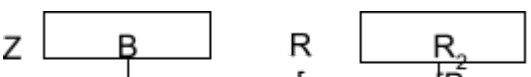
BB: BEGIN; ("realization C = E"); WRITE R1; END BB

Here, "end of computation" is the condition for terminating the computation. The condition must be specified when describing the DM.

Here, "realization D = B" is an assignment statement or a clause that performs the formation of each of the components of C using E. This can be just a subtraction, comparison and setting of some sign, or any expression generated by the application area and implemented by the language. With one execution of the loop body, one C component is calculated.

SAZ^2 is implemented using a cycle, an integral part of which are actions that provide access to each El-data or A-data, which is a component of the C-data. The loop construction is shown in the following example:

B is a simple C-data; C - simple El-data; B = { ∪C }; D - E-data is located in A-memory;



RG: R[B]; R^{μ} = < R_1[C] >;

R is placed on the A-tape, whose ID is A1. Between D and B there is a certain dependence, which is realized by the sentence “realization D = C”. The algorithmic construction consists of two parts.

READ A1

PERFORM BB (D_A_{A1} = “N”).

BB: BEGIN; (“realization D = C”); READ A1; END BB

Here, “realization D=C” is an assignment operator or statement that performs the processing or formation of D using C. The statement can contain any arithmetic or algebraic expression that contains C as an operand. Any expression in the sense of the possibilities that can be used in the practical implementation of DSA-theory as a means of describing decomposition schemes and related calculations. If this is an assignment operator, then it has such an expression on the right side, which, after the execution of the loop, should ensure the formation of the El-data D.

Preliminary description of the program structure

Frame body components. An arbitrary number of A-dependencies can be associated with each DPS node, which can be implemented both by simple sentences and complex ones. All A-dependencies that correspond to one DPS node were previously called ACN. All sentences of the algorithmic language that represent ACN are called a paragraph. The totality of all paragraphs is the program. The program can also be represented by a tree. Each node in this tree has one paragraph associated with it. Hereinafter, this tree is called the algorithm tree (AT). Both ACN and paragraph correspond to the main A-data.

As previously mentioned, one AT node can be associated with a DM and/or more (at least) one SAD^1. The cycle headers that implement DM and SAD^1 are replaced by a single header, since they all define the same cycle. All AC, which are the body of the cycle for DM and each SAD^1, are combined into one body. The connection between the paragraph with this heading and the paragraph with the corresponding merged loop body is the only connection between these paragraphs. This connection is called a hierarchical connection. Hierarchical connection is used to connect paragraphs. There can be only one hierarchical connection between two adjacent nodes. In the AT image, an edge connecting two parent and child nodes indicates that the paragraphs corresponding to these nodes will be joined by a hierarchical connection. The relationship between all paragraphs of the program is depicted by AT. In this tree, the nodes correspond to the paragraphs of the program.

The whole set of hierarchically articulated paragraphs is called a program. The program contains paragraphs, the number of which corresponds to the number of AT nodes. The paragraphs are joined by hierarchical connection in the same order as the corresponding DPS nodes are connected. That is, the DPS and AT trees are isomorphic.

There is a dependence between the syntactic means of the language, depicted in Fig. 5.

There are differences between the sections of the program depending on where the corresponding AT nodes are located.

1. Root node. There may be sentences here that implement cycle headers, SADs, and DMs. There cannot be loop bodies here.

2. End node. Sentences that implement the bodies of the SAD cycles and (or) DM of the parent node. There can be no loop headers here.

3. Intermediate node. There can be sentences here - cycle headers that implement their own SAB and (or) DM and cycle body sentences that implement the SAB and (or) DM of the parent node.

AAD can be implemented in paragraphs corresponding to any AT node.

The order in which the statements associated with a AT node are executed depends on several factors:

1. The content of the ACN.

2. The order of placement of A-fragments on the input A-tape.
3. The order of placement of A-fragments on the output A-tape.

ACN representation

The placement situations for the main A-data components are listed below. Considering that A-data is accessed by name within a record, the order in which D-data is placed in the record does not affect the order in which the SSPs that process this D-data are placed.

All sentences of the paragraph can be divided into groups. Each of the groups corresponds to one A-fragment. The connection of the group and the A-fragment is due to the fact that the A-fragment contains D-data, which are the initial for the A-dependencies implemented by the group's proposals. Hereinafter, this group will be called the AT-sentence.

The order of the AT-sentences in a paragraph must be the same as the order of the A fragments on the A tape.

A-fragments have an order relative to the boundaries of the area of the main A-data. If all A-fragments of the main A-data have certain places on the A-tape, then the corresponding AT-sentences must be written in the same order. AT-sentences will be concatenated by sequential concatenation. But a situation is possible when two or more AT-sentences are dependent and the order of their connection does not correspond to the order of the A-fragments on the A-tape. In this case the paragraph cannot be composed. In programming practice, they are looking for the opportunity to change the order of placement of A-fragments and place A-fragments according to the order of AT-sentences. Repeated passes along the A-tape can also be made.

A-fragments have a relative order. The situation where A-fragments have an order in relation to each other is similar to the placement with an order relative to the boundaries of the area. The order of the AT-sentences must be the same as the order of the corresponding A fragments. The connection of AT-sentences will be sequential. If the A-fragments are separated by unproductive A-fragments, then each AT-sentence must be conditional, and their connection must be alternative. The essence of the conditions is to recognize an unproductive A-fragment and skip it on the A-tape.

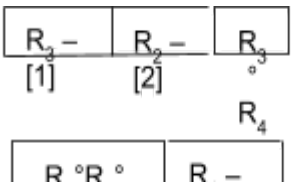
A situation is possible when two or more AT-sentences are dependent and the order of their articulation does not correspond to the order of A-fragments on the A-tape placed relative to each other. In this case, the paragraph cannot be composed. In programming practice, they are looking for the opportunity to change the order of placement of A-fragments and place A-fragments according to the order of AT-sentences. Repeated passes along the A-tape can also be made.

Joint placement of components of the first type (mix1). All A-fragments, as components of the main A-data, are mixed in the form of mix1. The corresponding AT-sentences are connected by an alternative connection. The conditions must check (distinguish) the encountered A-fragment and execute the corresponding AT-sentence. The entire paragraph is one alternative connection.

Joint placement of components of the second type (mix2). With this placement, the entire area of the main A-data is considered divided into zones. In the zones, A-fragments are combined with a relative placement order. These zones and the rest of the A-fragments are considered as a set of components on the A-tape, which are mixed like mix1. For components with placement mix1, an alternative connection of the corresponding AT-sentences is used. Within the zones where the components are placed relative to each other, serial connection is used with the refinements described above for such a case.

Joint placement of components of the third type (mix3). With this placement (Fig. 6), A-fragments can have certain positions at the beginning (a) or at the end (b) of the area occupied by the main A-data. Components of the main A-data, which have certain positions, can also be at the beginning and at the end of the area (c).

With this arrangement, the area is considered divided into zones. Zones of A-fragments with certain places and zones where A-fragments are mixed like mix1.



AT-sentences corresponding to A-fragments placed in a zone with certain positions are connected by a sequential connection. Let's call the resulting AK (temporarily, in the conditions of the example) A. The AT-sentences corresponding to the A-fragments placed in the zone with joint placement are connected by an alternative connection. Let's call the resulting AK B. The resulting structures (A and B) are connected by a sequential joint in the order that exists between the corresponding zones. Under the conditions of the examples in Fig. 6. connections will be as follows:

a: AC A + AC B

b: AC B + AC A

c: AC A + AC B + AC A

Any of these connections is a paragraph that corresponds to the main A-data.

Conditional A-fragments. Some A-fragments may be absent by the time of calculation. This situation may be due to the nature and values of P-properties and, accordingly, P-data. Information about their presence may be known by the beginning of the calculation or not. That a record or subarea is missing can only become known after all the components of the main A-data have been processed or the end of the subarea containing them has been reached. The existence of such components should be taken into account when articulating AC in the paragraph.

A-fragments with certain places can be conditional. The corresponding AT sentences must be conditional. The condition checks for the type encountered in the record or subarea. The algorithmic construction will be executed if the corresponding component is encountered. In a condition, D-data can be checked, to which there is access and which may contain a sign of the presence of a conditional D-data.

A situation is possible when an A-fragment with a certain place is a sign of the end of a subarea. If this component is conditional, then the next unconditional component must also be taken into account as a sign of the completion of the subarea.

Example (Fig.7.): If there are more than one conditional A-fragments (records A_2, A_3, ... A_n) with certain places that are a condition for completing the subarea, then they all must be taken into account in the condition that checks the sign of the completion of the sub-area. The condition for the completion of the subdomain A_1 will be the first of the encountered records A_2, A_3, ... A_n.

The presence of conditional components must be taken into account when arranging ACs

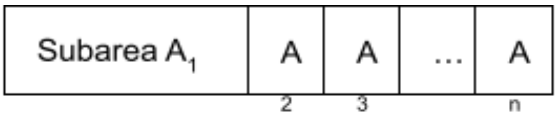


Fig. 7. Conditional A-fragments with certain places as signs of the completion of the subdomain.

generated by P-data that have a relative sequence order. As in the case of D-data with certain positions, articulated AKs must be supplemented by a condition for checking their presence. For verification, either A-data with a service El-data feature, or the presence of the following subarea is used. As with position-specific conditional A-fragments, if conditional components with a relative order are used to define the end of a subarea, then the subarea termination condition must take into account the conditional components.

Mandatory A-fragments. When placing the components of the main A-data, it is possible that some components must be present. When co-locating, a service El-data must be defined, which performs the function of a flag. The flag must be set to the “component missing” state before working with collocated components, and then during their processing, when the required component is encountered, the flag is reset to the “component present” state. If the component is not

met, then this is considered a specification error on the basis on which the DS was built. Each required component has its own flag associated with it.

Program Structure

All program offers can be divided into two categories. The first category includes proposals that directly implement the processing of A-dependencies. These sentences use the names of El-data (and, accordingly, El-properties). The second category includes sentences that implement SAD of both kinds, DM, access to D-data on the A-tape, conditional statements, the existence of which is due to the placement of D-data in areas and records on the A-tape. The totality of all sentences of the first category of a paragraph is called the functional core (FC) of this paragraph. The totality of all sentences of the second category of a paragraph is called an algorithmic frame (AF) of this paragraph.

Consider a DS whose DA has two nodes (Fig. 8.). The K_0-level paragraph should have a cycle heading that implements its own SAD. In an AT-sentence that corresponds to level K_1, there must be an operator that implements the body of this SAD (level K_0). At level K_0, there should also be a cycle header that will provide the implementation of the AAD calculation that correlates with level K_1. Also, there should be a loop header that will provide access to the records on the A-tape.

All three loops mentioned above are controlled by the same loop parameter, the number of entries. These three cycles are synthesized into one. The header is common to everyone, and the body of the loop will contain the following components:

AT-sentence of level K_1, which includes statements that implement the AAD of level K_1, and statements that implement the body of the SAD cycle of level K_0.

The order and nature of the connections of operators that implement the A-dependencies of the levels K_1 and K_0 depends on the placement of data in the areas and will be implemented by the corresponding connections in the corresponding AT-sentences. The number of records is unknown. A-tape ID A1.

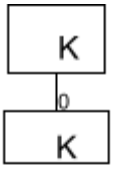


Fig. 8. Twolevel DS.

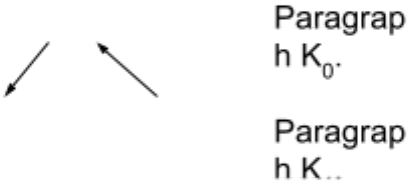


Fig. 9. Graphical representation programs

– algorithmic frame components

– functional core components

In paragraphs of level K_0, in the first and second cases, the statements BEGIN, READ A1, PERFORM, END AA are an algorithmic frame that provides an implementation of the SAD of level K_0. The algorithmic frame provides the implementation of the functionality of the K_0 level. In paragraphs of the K_1 level, in both cases, the statements BEGIN, READ A1, END BB are also an algorithmic frame that provides the implementation of the functionality of the K_1 level.

Paragraph corresponding to level K_0

AA: BEGIN
READ A1
Operator 1

PERFORM BB (D_A_{A1} = "N").

Operator K
END AA

Paragraph corresponding to level K_1

BB: BEGIN
Operator 1

Operator M
READ A1;
END BB

The set of algorithmic frames of all sections of the program, taking into account their joints and mutual placement, is called the algorithmic matrix (AM) of the program. The totality of all functional cores of all sections of the program, taking into account their articulations and mutual placement, is called the functional content (FC) of the program.

Graphically, the algorithm as a combination of the algorithmic matrix and functional content

is shown in fig. 9. Two arrows reflect the transfer of control from the header to the loop body. But then the hierarchical connection between paragraphs will be represented by a single line (but not by arrows). Hatching of the “brickwork” type depicts the components of algorithmic frames or an algorithmic matrix. The shading of the “wave” type depicts the components of the functional kernels or the functional content of the algorithm.

Additional remarks

The initial data for generating the algorithm is DS with tuples of node descriptions. From the point of view of a more accessible presentation, DS can be considered as a high-level abstraction algorithm scheme. In other words, a conventional flowchart, or other graphical representation of control movement or data flow, is built on top of the DS. Although from the point of view of real practical programming, this is not necessary. The result of the possible generation at the present level of development of the DSA-theory is the text of the program in the algorithmic language. This text is not intended for the practical work of a programmer with it. The text is an intermediate result and is considered as the initial data for subsequent compilation and creation of a working application program, and it is needed only during the testing of a possible generator of algorithms.

The natural desire of researchers in the field of programming is the desire to formalize the programming process. A program can be viewed as the result of some operations on operands. If we continue the search in this direction, then the concept of certain primitives, operations on them and structures made up of primitives will appear. This approach makes it possible to apply or develop a mathematical apparatus for the analysis, verification and research of programs. On this path, the ideas of structured programming were born [4]. Within the framework of the DSA-theory, the idea of “operation-operand” takes place both in relation to DS and in relation to algorithms written using an algorithmic language. The article describes operations and operands in relation to algorithms. Operations are sequential, conditional, alternative and hierarchical connections. The operands are AC and SSP, which implement DM and A-dependencies.

As algorithmically relevant factors of a different type are taken into account in the implementation of the CA, additional paragraph nodes appear in the AT and, as a result, the isomorphism of the DPS and AT trees is destroyed. This phenomenon will be described in the next article.

Conclusions

The article describes how to place P-data on the A-tape. Making changes to the CA, due to the methods of placing P-data, allows you to create a program text, which brings DS, as a theoretical model, closer to real applied algorithms.

Analysis and study of CA, presented as a text in an algorithmic language, allows us to define a number of concepts: functional core, algorithmic frame, functional content and algorithmic matrix. These concepts are necessary for further research and synthesis of algorithms.

DS, as a theoretical model focused on the development of real application programs, requires, using specific concepts, to describe some factors of a task or application area at an early stage, starting with research. These factors are CS, C-properties and some others. Such a description of the problem within the framework of the DSA-theory creates the prerequisites for the gradual abolition of the stages of algorithmization and programming, as the most intellectually saturated.

1. *Kolesnyk V.G.* DSA-theory as a prototype of the theory of applied algorithms // Problems in programming. – 2012. – № 1. P. 17-33. (In Russian)
2. *Robert Harper.* Existential Type. What, If Anything, Is A Declarative Language? 18 July 2013.
3. *Lloyd, J.W.* Practical Advantages of Declarative Programming. GULP-PRODE'94 1994 Joint Conference on Declarative Programming. Peniscola (Spain), September 19-22, 1994.
4. O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare *Structured Programming*, Academic Press, London, 1972. ISBN 0-12-200550-3.