Computer Science Journal of Moldova, vol.**24**, no.3(72), 2016

Proving Properties of Programs on Hierarchical Nominative Data*

Ievgen Ivanov, Mykola Nikitchenko,
Volodymyr G. Skobelev

Abstract

In the paper we develop methods for proving properties of programs on hierarchical nominative data on the basis of the composition-nominative approach. In accordance with this approach, the semantics of a program is a function on nominative data constructed from basic operations using compositions (operations on functions) which represent programming language constructs. Nominative data can be considered as a class of abstract data models which is able to represent many concrete types of structured and semistructured data that appear in programming. Thus, proofs of properties of programs depend on proofs of properties of compositions and basic operations on nominative data.

To simplify the parts of such proofs that deal with program compositions we propose to represent compositions of programs on nominative data using effective definitional schemes of H. Friedman. This permits us to consider proofs in data algebras (which are simpler to derive, automate, etc.) instead of proofs in program algebras. In particular, we demonstrate that the properties of programs related to structural transformations of data can be reduced to the data level. The obtained results can be used in software development and verification.

Keywords: Programming language semantics, algorithmic algebras, nominative data, composition, Friedman scheme.

* This work was supported in part by the project "Development of logic-algorithmic methods for investigation of formal models of natural languages" of Taras Shevchenko National University of Kyiv, Ukraine, Ref. Nr. 0116U004780

1 Introduction

The importance of the problem of elaborating the theory of programming and connecting it with software development practice was recognized by many researchers. In particular, it was mentioned as one of the grand challenges in computing by T. Hoare in his influential paper [1]. More generally, one may argue that development of tools and methods of program analysis that can make sure that it has the desired runtime properties before the program is run (e.g. model checking, verification against a formal specification using logical methods and automatic theorem provers, etc.) is a very important research topic.

In this paper we consider one aspect of the mentioned problem that is concerned with simplification of the process of proving properties of programs which operate on complex data structures (e.g. records, multidimensional arrays, trees, etc.). The types of properties we consider are the properties which can be described by special predicates on input and output data of a program: if i is an input data, o is the output of a program on the input i, then the property can be formulated as $P(i, o)$, where P is a predicate such that its truth domain is a transitive binary relation. Another kind of property which we consider is monotonicity of a program as a function from input data to output data with respect to some preorder relation on data (or, in particular, equivalence relation).

The ability to check such properties is useful in many cases. For example, in terms of such properties one can formulate the statement of partial correctness of a cyclic program in Floyd-Hoare logic [2], [3], the statement about correctness of implementation of a procedure/function/method with respect to its specification in contract programming [4], the statement about preservation of the structure or content of the data by a program (if a program is intended to perform a certain transformation of its input like optimization, translation, compilation, etc.).

Usually a proof of such a property of a program can be done by induction on the program structure and it ultimately reduces to a number of proofs of properties of similar type for the basic operations on data and for programming constructs (compositions – such as sequential execution, branching, cycle, etc.).

However, the complexity of such a proof can be lowered, if one is able to reduce the proofs of properties of different compositions to proofs of properties of operations on data. In this paper we propose a way of achieving such a reduction by representing compositions using effective definitional schemes of H. Friedman. The achieved reduction permits us to consider proofs in data algebras (which are simpler to derive, automate, etc.) instead of proofs in program algebras. Using this approach we demonstrate that the properties of programs related to structural transformations of data can be simplified by reducing them to the data level.

An informal description of how our approach works is given below. Consider the following version of the greatest common divisor computation algorithm (GCD algorithm).

Input: x, y (integer), local variables: a, b (integer)

```
a:=x; b:=y;
while a ≠ b do begin
if a>b do a:=a-b;
if b>a do b:=b-a;
end
```

The program has a state which can be represented as an association between variable names a, b, x, y and integer values e.g. its initial state can be $d = [x \mapsto 10, y \mapsto 5, a \mapsto 10, b \mapsto 5]$. Such an association can be formalized as a nominative data. We will denote the value associated with a name x in a data d as $d(x)$. The program state changes during execution, however, all operations which change the state (assignments $a := a - b$, $b := b - a$) leave the value $gcd(a, b)$ unchanged. Let us define as P the input-output relation of the program, i.e. the set of pairs (d_i, d_o) of nominative data such that if d_i is the initial state of the program, then d_o is the final state of the program. Let us denote as $\leqslant$ the following binary relation: if d_1, d_2 are nominative data which give values to the names a, b, x, y, then $d_1 \leqslant d_2$ if and only if

$$d_1(x) = d_2(x) \wedge d_1(y) = d_2(y) \wedge gcd(d_1(a), d_1(b)) = gcd(d_2(a), d_2(b)).$$

Obviously, $\leqslant$ is a transitive relation. If we can show $(d_i, d_o) \in P$ implies that $d_i \leqslant d_o$, then we can easily conclude that the program is partially correct, i.e. if the program terminates, then $a = b = gcd(x, y)$, so it indeed computes the greatest common divisor of x and y. Thus for proving the property of partial correctness of this program it is sufficient to show that its input-output mapping (P) is an increasing function in the sense of some transitive relation. Now our observation is that to prove that P defines an $\leqslant$-increasing function, it may be sufficient to check that all basic transformations of data (e.g. assignments) which appear in the program's source code are $\leqslant$-increasing without analyzing how these basic transformations are composed using various programming constructs like the conditional operator (if) and cycle operator (while).

The benefit of this approach to proving partial correctness (or other properties) of a program is that when this approach is applicable, it gives its user the ability to reuse the proof of a program's property when a program undergoes various changes/improvements/optimizations that do not change the set of basic operations (transformations) of data which appear in it. For example, if we modify the above mentioned version of the GCD algorithm in the following way:

Input: x, y (integer), local variables: a, b (integer)

```
a:=x; b:=y;
while a ≠ b do begin
while a>b do a:=a-b;
while b>a do b:=b-a;
end
```

then the modified algorithm still consists of the same basic operations as the original one, so it is still partially correct.

To describe formally our approach we need a formal model of complex data structures used in programming and of programs that operate on such data. We choose the formal models of data, programs and programming constructs provided by the composition-nominative approach [5]. In accordance with this approach, the denotational semantics of a program is a function on nominative data [5] (a class of abstract models of data which is able to represent many concrete types of structured and semistructured data that appear in programming) constructed from basic operations on nominative data using compositions (operations on functions) which represent programming language constructs. The model of nominative data is particularly suitable for our purposes since, it was demonstrated [11] that nominative data with complex names and/or values can adequately represent many data structures used in programming practice. For example, the data representable in JSON (JavaScript Object Notation) data-interchange format, which is very popular in web development, can be naturally modeled using nominative data.

Besides data formalization, we will need the formalizations of common programming language constructs in terms of operations on programs on nominative data. As such formalizations we will use the operations of the Associative Nominative Glushkov Algorithmic Algebra (ANGAA) introduced in [11] which is a rich, but tractable generalization of Glushkov algorithmic algebras [14] to programs on complex data structures.

We give the necessary preliminaries about the composition-nominative approach in the next section.

We will use the following notation:

- $f : A \to B$ denotes a total function from a set A to B;
- $f : A \tilde{\to} B$ denotes a partial function from a set A to B;
- $f(x) \downarrow$ means that a partial function f is defined on a value x;
- $f(x) \uparrow$ means that a partial function f is undefined on a value x;
- $\cong$ denotes the strong equality: $f(x) \cong g(x)$ means that either f and g are both defined on x and have the same value on x, or f and g are both undefined on x.

2 Composition-Nominative Approach

The composition-nominative approach [5] aims to propose a mathematical basis for development of formal methods of analysis and synthesis of software systems. According to this approach, program models are specified as *composition-nominative systems* (CNS) which consist of simpler systems: composition, description, and denotation systems.

Composition system defines semantic aspects of programs, description system defines syntactical aspects, and denotation system specifies meanings of descriptions. Semantics of programs are defined as partial functions over a class of data processed by programs and means of construction of complex programs from simpler programs (e.g. branching, cycle, etc.) are defined as n-ary operations (called compositions) over functions over data. A composition system can be specified as two algebras: data algebra and function algebra. Syntactically programs are represented as terms in the function algebra. The corresponding term algebra defines a descriptive system and the ordinary procedure of term interpretation gives a denotation system.

Data on which programs operate are modeled as *nominative data* [5]. Such data are special kinds of associations between names and values. There are several types of nominative data [8], [9], [10]. Among them the simplest type is the class of nominative sets, where a nominative set is a partial function from a set of abstract names to a set of abstract values [5], [8]. Nominative sets are frequently used in denotational semantics for formalizing program state [17]. In the general case, nominative data are classified in accordance with the following parameters:

- *values* can be simple (unstructured) or complex (structured),
- *names* can be simple (unstructured) or complex (structured).

Here “complex values” mean that the values corresponding to names in a nominative data can be nominative data themselves. Complex (structured) names are understood as strings consisting of simple (unstructured) names. The possible values of the mentioned parameters give four types of nominative data which are denoted as follows:
TND_{SS} – nominative data with simple names and simple values,
TND_{CS} – nominative data with complex names and simple values,
TND_{SC} – nominative data with simple names and complex values,
TND_{CC} – nominative data with complex names and complex values.

The formal definitions of the mentioned types of nominative data are given below.

- For any fixed sets of names V and values A, the class of data of the type TND_{SS} over V and A is defined as $D_0(V, A) = V \overset{n}{\to} A$, where $V \overset{n}{\to} A$ denotes the set of partial functions from V to A which have a finite graph. The elements of this class are denoted using notation $[v_1 \mapsto a_1, ..., v_n \mapsto a_n]$, where $v_i \in V$ are names and $a_i \in A$ are the corresponding values. For example, data $d = [u \mapsto 1, v \mapsto 2]$ belongs to $D_0(V, A)$, where $u, v \in V$ are distinct elements and $\{1, 2\} \subseteq A$, $dom(d) = \{u, v\}$ and $d(u) = 1$, $d(v) = 2$.

- For any fixed sets of names V and values A, the class of data of the type TND_{SC} over V and A is $D_1(V, A) = ND(V, A)$, where

  – $ND(V, A) = \bigcup_{k \geq 0} ND_k(V, A)$,

  – $ND_0(V, A) = A \cup \{\emptyset\}$,

  – $ND_{k+1}(V, A) = A \cup \left(V \overset{n}{\longrightarrow} ND_k(V, A)\right), \quad k \geq 0.$

  Here, we denote by $\emptyset$ the empty nominative data, i.e. a function with an empty graph (this notation is also used for the empty set).

  Data of type TND_{SC} are hierarchically constructed. An example of such data is $[u \mapsto 1, v \mapsto [w \mapsto 2]]$, where $u, v, w \in V$, $1, 2 \in A$. Such data can be represented by oriented trees (of varying arity) with arcs labelled by names and with leafs labelled by elements from A or $\emptyset$.

  A *path* is a nonempty finite sequence $(v_1, v_2, ..., v_k)$, $v_1, ..., v_k \in V$.

  For a given data d, a *value of a path* $(v_1, v_2, ..., v_k)$ in d is defined by the expression $d(v_1, v_2, ..., v_k) \cong (...((d(v_1))(v_2))...(v_k))$.

  We say that a path $(v_1, v_2, ..., v_k)$ is *a path in a data* $d \in ND(V, A)$, if a value of $(v_1, v_2, ..., v_k)$ in d is defined, i.e. $d(v_1, v_2, ..., v_k) \downarrow$ (a path in data corresponds to a path from the root to a node in an oriented tree). A *terminal path* in a data $d \in ND(V, A)$ is a path in d such that its value belongs to $A \cup \{\emptyset\}$. The least k such that $d \in ND_k(V, A)$ is the *rank* of a data d.

- For any fixed sets of names V and values A, the class of data of the type TND_{CS} over V and A is defined as $D_2(V, A) = NDVS(V, A)$, where $NDVS(V, A)$ is the set of all elements of $A \cup (V^+ \xrightarrow{n} A)$ such that either $d \in A$, or $d \in V^+ \xrightarrow{n} A$ and all strings from $dom(d)$ are pairwise incomparable in the sense of the prefix relation (*principle of unambiguous associative naming*). An example of such data is $[uv \mapsto 1, uw \mapsto 2, w \mapsto 3]$, $u, v, w \in V$. Such data have *complex names* i.e. names that are strings.

- For any fixed sets of names V and values A, the class of data of the type TND_{CC} over V and A is defined as $D_3(V, A) = NDVC(V, A)$, where $NDVC(V, A)$ is the class of all data $d \in ND(V^+, A)$ such that for any two paths $(u_1, u_2, ..., u_k)$ and $(v_1, v_2, ..., v_l)$ in d, neither of which is a prefix of another, the words $u_1u_2...u_k$ and $v_1v_2...v_l$ are incomparable in the sense of the prefix relation (*principle of unambiguous associative naming*). Such data is also called *complex-named data* [10]. An example of such data is $[uv \mapsto 1, w \mapsto [uw \mapsto \emptyset]]$, $u, v, w \in V$.

3 Basic Operations on Nominative Data

The basic operations on nominative data are the operations of
– *denaming* (taking the value of a name),
– *naming* (assigning a new value to a name),
– *overlapping.*

Let us define these operations for data of the most interesting and complex type TND_{CC}.

Let V and A be fixed sets of names and basic values respectively.

Definition 1 (Denaming)**.** *The (associative) denaming is an operation $v \Rightarrow_a$ with a parameter $v \in V^+$ defined by induction on the length of v:*

- *if $v \in V$, then* $v \Rightarrow_a (d) \cong \begin{cases} d(v), & \text{if } d(v)\downarrow; \\ d/v, & \text{if } d(v)\uparrow \text{ and } d/v \neq \emptyset; \\ \text{undefined}, & \text{if } d(v)\uparrow \text{ and } d/v = \emptyset, \end{cases}$
  where $d/u = [v_1 \mapsto d(z) \mid d(z)\downarrow, z = uv_1, v_1 \in V^+]$;

- *if $v \in V^+ \backslash V$, then $v \Rightarrow_a (d) \cong v_2 \Rightarrow_a (v_1 \Rightarrow_a (d))$, where v_1 is the first symbol of v and v_2 is the suffix, i.e. v_1, v_2 are (unique) words such that $v = v_1 v_2$ and $v_1 \in V$.*

The following examples illustrate this operation:

- $u \Rightarrow_a ([u \mapsto 1, v \mapsto 2]) = 1$;
- $(uv) \Rightarrow_a ([u \mapsto [vw \mapsto 1, u \mapsto 2]]) = [w \mapsto 1]$.

This operation has the following property (*associativity*) [10]:

$$u \Rightarrow_a (d) \cong u_n \Rightarrow_a (u_{n-1} \Rightarrow_a (... u_1 \Rightarrow_a (d)...))$$

for all complex names $u, u_1, u_2, ..., u_n \in V^+$ such that $u = u_1 u_2 ... u_n$.

Definition 2 (Naming)**.** *Naming is an unary operation $\Rightarrow v$ with a parameter $v \in V^+$ such that $\Rightarrow v(d) = [v \mapsto d]$.*

Overlapping is a kind of updating operation which updates the values of names in its first argument with the values of names in its second argument. For different types of nominative data different overlapping operations can be considered. We will define two kinds of overlapping: global and local overlapping. Global (associative or structural) overlapping ∇_a updates several values in the first argument while the local one ∇_a^v (with a parameter name v) updates only one value which is associated with the name v.

Global overlapping can be used, e.g. for formalizing procedures calls, while the local overlapping can be used as a formalization of the assignment operator in programming languages.

Definition 3 (Global overlapping)**.** *For nominative data of the type TND_{CC}, global overlapping is a binary operation ∇_a defined inductively by the rank of the first argument as follows.*

Let $NDVC_k(V, A) = NDVC(V, A) \cap ND_k(V^+, A)$ be the data from the set $NDVC(V, A)$, the rank of which is $\leq k$.

Induction base of the definition. If $d_1 \in NDVC_0(V, A)$, then

$$d_1 \nabla_a d_2 \cong \begin{cases} d_2, & \textit{if } d_1 = \emptyset \textit{ and } d_2 \in NDVC(V, A) \backslash A; \\ \textit{undefined}, & \textit{if } d_1 \in A \textit{ or } d_2 \in A. \end{cases}$$

Induction step of the definition. Assume that the value $d_1 \nabla_a d_2$ is already defined for all d_1, d_2 such that $d_1 \in NDVC_k(V, A)$. Let

$$d_1 \in NDVC_{k+1}(V, A) \backslash NDVC_k(V, A).$$

Then $d_1 \nabla_a d_2 = d$, where d is defined for each name $u \in V^+$ as follows:

1) $d(u) = d_2(u)$, if $u \in dom(d_2)$ and u does not have a proper prefix which belongs to $dom(d_1)$;

2) $d(u) = d_1(u) \nabla_a (d_2/u)$, if $d_1(u)$ is defined and does not belong to A and u is a proper prefix of some element of $dom(d_2)$, where $d_2/u = [v_1 \mapsto d_2(v) \mid d_2(v) \downarrow, v = uv_1, v_1 \in V^+]$;

3) $d(u) = d_2/u$, if $d_1(u)$ is defined and belongs to A and u is a proper prefix of some element of $dom(d_2)$;

4) $d(u) = d_1(u)$, if $d_1(u)$ is defined and u is not comparable (in the sense of the prefix relation) with any element of $dom(d_2)$;

5) $d(u) \uparrow$, otherwise.

The global overlapping has the following properties [10]:

- $[u \mapsto d_1] \nabla_a [v \mapsto d_2] = [u \mapsto d_1, v \mapsto d_2], \quad u, v \in V, \quad u \neq v$;
- $[uv \mapsto d_1] \nabla_a [u \mapsto d_2] = [u \mapsto d_2]$, $u, v \in V^+$, i.e. the value under a name u in the second argument overwrites the values under names in the first argument which are extensions of u;
- $[u \mapsto d_1] \nabla_a [uv \mapsto d_2] = [u \mapsto (d_1 \nabla_a [v \mapsto d_2])]$, if $u, v \in V^+$, $d_1 \notin A$, i.e. the value under a name uv in the second argument modifies values under prefixes of uv in the first argument.

Definition 4 (Local overlapping)**.** *For nominative data of the type TND_{CC} local overlapping is a binary operation ∇_a^v with a parameter $v \in V^+$ defined as follows: $d_1 \nabla_a^v d_2 \cong d_1 \nabla_a (\Rightarrow v(d_2))$.*

Definition 5. *Name checking predicate $u!$ on $NDVC(V, A)$ with a parameter $u \in V^+$ is defined as follows:*

$u!(d) = T$, if $u \Rightarrow_a (d) \downarrow$; $u!(d) = F$, if $u \Rightarrow_a (d) \uparrow$.

Definition 6. *Emptiness checking predicate $IsEmpty$ on $NDVC(V, A)$ is defined as follows:*

$IsEmpty(d) = T$, if $d = \emptyset$; $IsEmpty(d) = F$, if $d \neq \emptyset$.

Using the basic operations on nominative data, we can define an algebraic structure of nominative data.

Definition 7. *An algebraic structure of nominative data of the type TND_{CC} is defined as follows:*

$$NDAS_{CC}(V,A) = (NDVC(V,A); \emptyset, \{v \Rightarrow_a\}_{v\in V^+},$$
$$\{\Rightarrow v\}_{v\in V^+}, \{\nabla_a^v\}_{v\in V^+}, \{v!\}_{v\in V^+}, IsEmpty),$$

where $\emptyset$ is a constant – the empty nominative data.

Note that $NDAS_{CC}(V,A)$ is a structure without equality.

One can extend $NDAS_{CC}(V,A)$ with additional unary predicates and operations on nominative data.

Definition 8. *Let $k,l \in \mathbb{N}\cup\{0\}$, $p_1,p_2,...,p_k$ be partial predicates on $NDVC(V,A)$ and $f_1,...,f_l$ be partial functions of the type $NDVC(V,A)\tilde{\to}NDVC(V,A)$. An extended algebraic structure of nominative data of the type TND_{CC} with additional unary predicates $p_1,...p_k$ and operations $f_1,...,f_l$ is defined as follows:*

$$NDAS_{CC}(V,A;p_1,...,p_k;f_1,...,f_l) = (NDVC(V,A);$$
$$\emptyset, \{v \Rightarrow_a\}_{v\in V^+}, \{\Rightarrow v\}_{v\in V^+}, \{\nabla_a^v\}_{v\in V^+}, \{v!\}_{v\in V^+}, IsEmpty,$$
$$p_1,...,p_k,f_1,...,f_l),$$

where $\emptyset$ is a constant – the empty nominative data.

Definition 9. *A path in $d \in NDVC(V,A)$ is a nonempty sequence $(v_1,v_2,...,v_n)$ of words from V^+ such that the value $((d(v_1))(v_2)...)(v_n)$ is defined. This value $((d(v_1))(v_2)...)(v_n)$ is called the value of the path $(v_1,v_2,...,v_n)$ in d. A path is called a terminal path in d, if its value in d belongs to $A\cup\{\emptyset\}$.*

Definition 10. *1) $d_1 \in NDVC(V,A)$ is nominatively included in $d_2 \in NDVC(V,A)$, if either $d_1,d_2 \in A$ and $d_1 = d_2$, or $d_1,d_2 \notin A$ and for each terminal path $(v_1,v_2,...,v_n)$ in d_1 there is a terminal path $(v'_1,v'_2,...,v'_m)$ in d_2 such that $v_1v_2...v_n = v'_1v'_2...v'_m$ and the values of $(v_1,v_2,...,v_n)$ in d_1 and $(v'_1,v'_2,...,v'_m)$ in d_2 coincide.*

2) d_1,d_2 are nominative equivalent ($d_1 \approx d_2$), if d_1 is nominatively included in d_2 and d_2 is nominatively included in d_1.

4 Associative Nominative Glushkov Algorithmic Algebra

Programs on nominative data can be formalized as functions from nominative data (input data) to nominative data (output data) which can be constructed from the operations of the algebraic structure $NDAS_{CC}(V, A)$ using compositions which represent programming language constructs, e.g. sequential execution, branching, cycle, etc.

The set of such programs together with compositions forms an algorithmic algebra similar to, e.g. Glushkov algorithmic algebras [14].

In [11] the authors of this paper proposed a generalization of Glushkov algorithmic algebras to algebras of functions and predicates over nominative data of the type TND_{CC} (i.e. data with complex names and complex values) in order to obtain a rich, but tractable formal language for specifying and reasoning about programs. This generalization is called an Associative Nominative Glushkov Algorithmic Algebra (ANGAA).

Let V and A be fixed sets of basic names and values. Denote
$Pr_{CC}(V, A) = NDVC(V, A) \tilde{\to} \{T, F\}$,
$Fn_{CC}(V, A) = NDVC(V, A) \tilde{\to} NDVC(V, A)$.

We will assume that T and F do not belong to $NDVC(V, A)$.

We will call the elements of $Pr_{CC}(V, A)$ *(partial nominative) predicates* and the elements of $Fn_{CC}(V, A)$ *(partial binominative) functions*.

Let us denote by $\bar{U}$ the set of all tuples $(u_1, u_2, ..., u_n)$, $n \geq 1$ of complex names from V^+ such that whenever $i \neq j$, u_i and u_j are incomparable in the sense of the prefix relation.

- Sequential composition of functions (denoted using the infix notation) $\bullet : Fn(V, A) \times Fn(V, A) \to Fn(V, A)$ is defined as follows: for all $f, g \in Fn(V, A)$ and data d: $(f \bullet g)(d) \cong g(f(d))$.

- Prediction composition [14] $\cdot : Fn(V, A) \times Pr(V, A) \to Pr(V, A)$ is defined as follows: for all $f \in Fn(V, A)$, $p \in Pr(V, A)$, and data d: $(f \cdot p)(d) \cong p(f(d))$.

- Assignment composition $Asg^u : Fn(V,A) \to Fn(V,A)$ with a parameter $u \in V^+$ is defined as follows: for each $f \in Fn(V,A)$ and data d, $(As^u(f))(d) \cong d\nabla_a^u f(d)$.
- The composition of superposition into a function
$$S_F^{u_1,u_2,...,u_n} : Fn(V,A) \times (Fn(V,A))^n \to Fn(V,A)$$
with parameters $n \geq 1$ and $u_1,...,u_n \in V^+$ such that $(u_1,...,u_n) \in \bar{U}$ is defined as follows:
$$S_F^{u_1,...,u_n}(f, f_1, ..., f_n)(d) \cong f(...(d\nabla_a^{u_1} f_1(d))...\nabla_a^{u_n} f_n(d))...).$$
We will also use the following notation for this composition: for each tuple $\bar{u} = (u_1, u_2, ..., u_n) \in \bar{U}$, $S_F^{\bar{u}}$ denotes $S_F^{u_1,u_2,...,u_n}$.
- The composition of superposition into a predicate
$$S_P^{u_1,u_2,...,u_n} : Pr(V,A) \times (Fn(V,A))^n \to Pr(V,A)$$
with parameters $n \geq 1$ and $u_1,...,u_n \in V^+$ such that $(u_1,...,u_n) \in \bar{U}$ is defined as follows:
$$S_P^{u_1,...,u_n}(p, f_1, ..., f_n)(d) \cong p(...(d\nabla_a^{u_1} f_1(d))...\nabla_a^{u_n} f_n(d))...).$$
We will also use the following notation for this composition: for each tuple $\bar{u} = (u_1, u_2, ..., u_n) \in \bar{U}$, $S_P^{\bar{u}}$ denotes $S_P^{u_1,u_2,...,u_n}$.
- Branching composition $IF : Pr(V,A) \times Fn(V,A) \times Fn(V,A) \to Fn(V,A)$ is defined as follows: for each $p \in Pr(V,A)$, $f, g \in Fn(V,A)$:

  $IF(p,f,g)(d) \cong f(d)$, if $p(d) \downarrow = T$.

  $IF(p,f,g)(d) \cong g(d)$, if $p(d) \downarrow = F$.

  $IF(p,f,g)(d)$ undefined, if $p(d) \uparrow$.
- Cycle composition $WH : Pr(V,A) \times Fn(V,A) \to Fn(V,A)$ is defined as follows: for each $p \in Pr(V,A)$, $f \in Fn(V,A)$, and d:

$WH(p,f)(d)\downarrow = f^{(n)}(d)$, if there exists $n \geq 0$ such that $(f^{(i)} \cdot p)(d)\downarrow = T$ for all $i \in \{0,1,...,n-1\}$ and $(f^{(n)} \cdot p)(d)\downarrow = F$, where $f^{(n)}$ is a n-times sequential composition of f with itself ($f^{(0)}$ is the identity function), and $WH(p,f)(d)$ is undefined otherwise.

- Negation $\neg : Pr(V,A) \to Pr(V,A)$ is a composition such that for each $p \in Pr(V,A)$ and data d: $(\neg p)(d) \cong T$, if $p(d)\downarrow = F$; $(\neg p)(d) \cong F$, if $p(d)\downarrow = T$; $(\neg p)(d)$ is undefined, if $p(d)\uparrow$.

- Disjunction $\vee : Pr(V,A) \times Pr(V,A) \to Pr(V,A)$ is a composition defined as follows: for each $p_1, p_2 \in Pr(V,A)$ and data d:

$$(p_1 \vee p_2)(d) \cong \begin{cases} T, & \text{if } p_1(d)\downarrow = T \text{ or } p_2(d)\downarrow = T; \\ F, & \text{if } p_1(d)\downarrow = F \text{ and } p_2(d)\downarrow = F; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

- Identity composition $Id : Fn(V,A) \to Fn(V,A)$ is defined as follows: $Id(f) = f$ for all $f \in Fn(V,A)$.

- True constant predicate (null-ary composition) $True \in Pr(V,A)$ is defined as follows: $True(d)\downarrow = T$ for all data d.

- Bottom function (null-ary composition) $\bot_F \in Fn(V,A)$ is defined as follows: $\bot_F\ (d)\uparrow$ for all data d.

- Bottom predicate (null-ary composition) $\bot_P \in Pr(V,A)$ is defined as follows: $\bot_P\ (d)\uparrow$ for all data d.

- Name checking predicate (null-ary composition) with a parameter $u \in V^+$: $u!(d) = T$, if $u \Rightarrow_a (d)\downarrow$; $u!(d) = F$, if $u \Rightarrow_a (d)\uparrow$.

- Empty constant function (null-ary composition): $Empty(d) = \emptyset$.

- Emptiness checking predicate (null-ary composition):
  $IsEmpty(d) = T$, if $d = \emptyset$; $IsEmpty(d) = F$, if $d \neq \emptyset$.

Our generalization of Glushkov algorithmic algebras to an algebra of programs on hierarchical data is defined below.

Definition 11. *An Associative Nominative Glushkov Algorithmic Algebra (ANGAA) is a two-sorted algebra*

$$NGA^a_{CC}(V,A) = (Pr_{CC}(V,A), Fn_{CC}(V,A); \bullet, IF, WH, \cdot, \{Asg^u\}_{u\in V^+},$$
$$\{S_F^{\bar{u}}\}_{\bar{u}\in\bar{U}}, \{S_P^{\bar{u}}\}_{\bar{u}\in\bar{U}}, \vee, \neg, Id, True, \bot_F, \bot_P, \{u!\}_{u\in V^+},$$
$$Empty, IsEmpty)$$

One can further extend ANGAA by adding constant symbols which denote certain fixed predicates from $Pr_{CC}(V,A)$ and/or functions from $Fn_{CC}(V,A)$ to its signature.

Definition 12. *Let* $k,l \in \mathbb{N}\cup\{0\}$*,* $p_1,p_2,...,p_k \in Pr_{CC}(V,A)$ *and* $f_1,f_2,...,f_l \in Fn_{CC}(V,A)$*.*

An extended Associative Nominative Glushkov Algorithmic Algebra (eANGAA) with predicate constants $p_1,...,p_k$ *and function constants* $f_1,...,f_l$ *is a two-sorted algebra*

$$NGA^a_{CC}(V,A;p_1,...,p_k;f_1,...f_l) =$$
$$(Pr_{CC}(V,A), Fn_{CC}(V,A); \bullet, IF, WH, \cdot, \{Asg^u\}_{u\in V^+},$$
$$\{S_F^{\bar{u}}\}_{\bar{u}\in\bar{U}}, \{S_P^{\bar{u}}\}_{\bar{u}\in\bar{U}}, \vee, \neg, Id, True, \bot_F, \bot_P, \{u!\}_{u\in V^+}, Empty, IsEmpty,$$
$$p_1,p_2,...,p_k,f_1,f_2,...f_l).$$

5 Effective Definitional Schemes and Generalization of eds Definability

The generalized recursion theory as proposed by H. Friedman [15] and subsequently developed in [16] investigates generalized notions of computability on objects of algebraic structures. In this context in [15] H. Friedman defined the notion of a generalized Turing algorithm and the equivalent notion of an effective definitional scheme (eds) [15]. Basically, eds are definitions by infinite cases which have a recursive enumerable structure. They can be used to give a very general definition of a computable function; in fact it was argued [16] that for reasonable definitions of computable functions over algebraic structures computable functions need to be eds definable.

Such a definition of a computable function can be described as follows [16]. Consider a language L with finitely many constant, relation, operation symbols interpreted in an algebraic structure M with some domain, constants, relations and operations. Then a function f (on the domain of M) is eds definable, if there is a set S of conditions of the form $\varphi_i(v, v_1, ..., v_n) \to t_i(v, v_1, ..., v_n)$ (eds) consisting of terms $t_i(v, v_1, ..., v_n)$ in L and basic semialgebraic conditions (i.e. finite conjunctions of atomic formulas and their negations) $\varphi_i(v, v_1, ..., v_n)$ in L, where $v, v_1, ..., v_n$ are formal variable names, such that S is effective (recursively enumerable as a set of strings) and there exist elements $a_1, ..., a_n$ of the domain of M (parameters of the definition) such that for each i: $f(x) = t_i(x, a_1, ..., a_n)$, if $\varphi_i(x, a_1, ..., a_n)$.

It is known [16, Theorem 2] that in a suitable formalization, programs expressible in imperative programming languages with variables ranging over M and assignments, stacks of values from the domain of M with the operations *Push* and *Pop,* conditional operators (*If-Then-Else*), and jumps (*Goto*) define eds definable functions.

However, as it is, the notion of eds definability has a limited applicability to semantics of programming languages, since it tells only what are computable functions from M (or more generally, M^n) to M or M^m, where the elements of M are considered as unstructured data.

In contrast, in the context of semantics of programming languages [17], [2], [3], it is more important to describe computability of the steps taken by the program during execution, which are usually transformations of structured program states to structured program states.

Below we generalize eds definability of functions on a structure M to eds definability of transformations of program execution states for programs operating on complex data structures (e.g. multidimensional arrays, lists, trees and tree-like structures, etc.) over M.

Let $V = \{v_1, v_2, ..., v_m\}$ be a fixed finite set of basic names and A be a fixed set of basic values.

Let $p_1, ..., p_k \in Pr_{CC}(V, A)$, $f_1, ..., f_l \in Fn_{CC}(V, A)$ be fixed finite sequences of predicates and functions. If $x_1, ..., x_n$ are variable names, denote by $T_{x_1,x_2,...,x_n}(V; p_1, ..., p_k; f_1, ..., f_l)$ the set of all terms in $NDAS_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l)$ in $x_1, ..., x_n$, and by $\Phi_{x_1,...,x_n}(V; p_1, ..., p_k; f_1, ..., f_l)$ the set of all basic semalgebraic conditions, i.e. formulas which have a form of a finite conjunction of atomic formulas in $NDAS_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l)$ or their negations (note that equality is not allowed).

For each term t in $T_{x_1,x_2,...,x_n}(V; p_1, ..., p_k; f_1, ..., f_l)$ or formula φ in $\Phi_{x_1,...,x_n}(V; p_1, ..., p_k; f_1, ..., f_l)$, denote by $[t]$ and $[\varphi]$ their standard interpretations (i.e. the corresponding partial function and predicate on tuples of elements of $NDVC(V, A)$).

Definition 13. *A function $f \in Fn_{CC}(V, A)$ is eds definable with respect to $p_1, ..., p_k$ and $f_1, ..., f_l$, if there exists a natural number n, data $d_1, d_2, ..., d_n \in NDVC(V, A)$, and a finite or countable set S of pairs of the form*

$\{(\varphi_i(x, x_1, x_2, ..., x_n), t_i(x, x_1, ..., x_n)) \mid i \in I\}$

(I is a set of indices $I = \mathbb{N}$ or $I = \{1, 2, ..., K\}$ for some natural K), where $\varphi_i(x, x_1, ..., x_n) \in \Phi_{x,x_1,...,x_n}(V; p_1, ..., p_k; f_1, ..., f_l)$ and $t_i(x, x_1, ..., x_n) \in T_{x,x_1,...,x_n}(V; p_1, ..., p_k; f_1, ..., f_l)$ and $x, x_1, ..., x_n$ are different variable names, such that

1) the set of all strings of the form $\varphi(x, x_1, ..., x_n) \to t(x, x_1, ..., x_n)$ for $(\varphi(x, x_1, ..., x_n), t(x, x_1, ..., x_n)) \in S$ in the alphabet $\{v_1, v_2, ..., v_m, \mathbf{,}, (,), x, x_1, ..., x_n, \emptyset, \Rightarrow, a, !, IsEmpty, \neg, \wedge, p_1, ..., p_k, f_1, ..., f_l\}$ is recursively enumerable (it is assumed that symbols with sub/superscripts ∇_a^v in terms are represented as ∇av).

2) For each $d \in NDVC(V, A)$ and $i \in I$, if $[\varphi_i](d, d_1, ..., d_n) \downarrow= T$, then $f(d) \cong [t_i](d, d_1, ..., d_n)$.

3) For each $d \in NDVC(V, A)$, if $[\varphi_i](d, d_1, ..., d_n) \uparrow$ for all $i \in I$, then $f(d) \uparrow$.

Note that the problem of checking whether a given set of the form $\{(\varphi_i(x, x_1, x_2, ..., x_n), t_i(x, x_1, ..., x_n)) \mid i \in I\}$ defines an eds definable function as in Definition 13 may be algorithmically undecidable. However, this does not have any implications on applicability of the notion of eds definable functions to the problems considered in this paper.

Definition 14. *A predicate $p \in Pr_{CC}(V,A)$ is eds definable with respect to $p_1,...,p_k$ and $f_1,...,f_l$, if there exists a natural number n, data $d_1,d_2,...,d_n \in NDVC(V,A)$, and a finite or countable set S of pairs of the form*

$\{(\varphi_i(x,x_1,x_2,...,x_n),b_i) \mid i \in I\}$

(I is a set of indices $I = \mathbb{N}$ or $I = \{1,2,...,K\}$ for a natural K, b_i is a Boolean value), where $\varphi_i(x,x_1,...,x_n) \in \Phi_{x,x_1,...,x_n}(V;p_1,...,p_k;f_1,...,f_l)$ and $b_i \in \{T,F\}$ and $x,x_1,...,x_n$ are different variable names, such that

1) the set of all strings of the form $\varphi(x,x_1,...,x_n) \to b$ for $(\varphi(x,x_1,...,x_n),b) \in S$ in the alphabet $\{v_1,v_2,...,v_m, \mathbf{,}, (,), x, x_1,...,x_n, \emptyset, \Rightarrow, a, !, IsEmpty, \neg, \wedge, p_1,...,p_k, f_1,...,f_l\}$ is recursively enumerable (it is assumed that symbols with superscripts ∇_a^v in terms are represented as ∇av).

2) For each $d \in NDVC(V,A)$ and $i \in I$, if $[\varphi_i](d,d_1,...,d_n) \downarrow = T$, then $p(d) \cong b_i$.

3) For each $d \in NDVC(V,A)$, if $[\varphi_i](d,d_1,...,d_n) \uparrow$ for all $i \in I$, then $p(d) \uparrow$.

6 Main results

Let us introduce the following notation.

- $PrEds_{CC}(V,A;p_1,...,p_k;f_1,...,f_l)$ is the set of all predicates in $Pr_{CC}(V,A)$ which are eds definable with respect to $p_1,...,p_k$ and $f_1,...,f_l$.
- $FnEds_{CC}(V,A;p_1,...,p_k;f_1,...,f_l)$ is the set of all functions in $Fn_{CC}(V,A)$ which are eds definable with respect to $p_1,...,p_k$ and $f_1,...,f_l$.

The following theorem shows that all programs of eANGAA with predicate constants $p_1,...,p_k$ and function constants $f_1,...,f_l$ are eds definable with respect to $p_1,...,p_k$ and $f_1,...,f_l$.

Theorem 1 (eds definability of programs of eANGAA)**.** *The sets $PrEds_{CC}(V,A;p_1,...,p_k;f_1,...,f_l)$, $FnEds_{CC}(V,A;p_1,...,p_k;f_1,...,f_l)$ form a subalgebra of $NGA^a_{CC}(V,A;p_1,...,p_k;f_1,...,f_l)$.*

Proof. (Sketch)

It is easy to check that all null-ary compositions of the algebra $NGA^a_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l)$ are eds definable by Definition 13 and Definition 14.

The fact that all unary and binary compositions of the algebra $NGA^a_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l)$ (sequential composition, branching, cycle, etc.) preserve eds definability can be proven similarly to [16]. This implies the statement of the theorem. □

For any transitive binary relation $\leqslant$ on $NDVC(V, A)$ let us denote:

- $PrM_{CC}(V, A, \leqslant)$ is the set of all $p \in Pr_{CC}(V, A)$ such that for all d_1, d_2, if $p(d_1)\downarrow$ and $d_1 \leqslant d_2$, then $p(d_2)\downarrow= p(d_1)$. The elements of $PrM_{CC}(V, A, \leqslant)$ are called $\leqslant$ -equitone predicates.

- $FnI_{CC}(V, A, \leqslant)$ is the set of all $f \in Fn_{CC}(V, A)$ such that for each d, if $f(d)\downarrow$, then $d \leqslant f(d)$. The elements of $FnI_{CC}(V, A, \leqslant)$ are called $\leqslant$-increasing functions.

- $FnM_{CC}(V, A, \leqslant)$ is the set of all $f \in Fn_{CC}(V, A)$ such that for each d_1, d_2, if $f(d_1)\downarrow$ and $d_1 \leqslant d_2$, then $f(d_2)\downarrow$ and $f(d_1) \leqslant f(d_2)$. The elements of $FnM_{CC}(V, A, \leqslant)$ are called $\leqslant$-monotone functions.

- $PrM^n_{CC}(V, A, \leqslant)$ is the set of all $p \in Pr^n_{CC}(V, A)$ such that for all $d_1, d_2, ..., d_n, d'_1, d'_2, ..., d'_n$, if $p(d_1, d_2, ..., d_n)\downarrow$ and $d_1 \leqslant d'_1$, $d_2 \leqslant d'_2$, ..., $d_n \leqslant d'_n$, then $p(d'_1, d'_2, ..., d'_n)\downarrow= p(d_1, d_2, ..., d_n)$. The elements of $FnM^n_{CC}(V, A, \leqslant)$ are called $\leqslant$-equitone n-ary predicates.

- $FnI^n_{CC}(V, A, \leqslant)$ is the set of all $f \in Fn^n_{CC}(V, A)$ such that for each $d_1, d_2, ..., d_n$, if $f(d_1, d_2, ..., d_n)\downarrow$, then $d_i \leqslant f(d_1, d_2, ..., d_n)$ for each $i = 1, 2, ..., n$. The elements of $FnM_{CC}(V, A, \leqslant)$ are called $\leqslant$-increasing n-ary functions.

- $FnM^n_{CC}(V, A, \leqslant)$ is the set of all $f \in Fn^n_{CC}(V, A)$ such that for each $d_1, d_2, ..., d_n, d'_1, d'_2, ..., d'_n$, if $f(d_1, d_2, ..., d_n)\downarrow$ and $d_1 \leqslant d'_1$, $d_2 \leqslant d'_2$, ..., $d_n \leqslant d'_n$, then $f(d'_1, d'_2, ..., d'_n)\downarrow$ and $f(d_1, d_2, ..., d_n) \leqslant$

$f(d'_1, d'_2, ..., d'_n)$. The elements of $FnM^n_{CC}(V, A, \leqslant)$ are called $\leqslant$-monotone n-ary functions.

Lemma 1. *Let $\leqslant$ be a transitive binary relation on $NDVC(V, A)$. Assume that:*

$f_1, f_2, ..., f_l, \Rightarrow u, u \Rightarrow_a, \nabla^u_a \in FnI_{CC}(V, A, \leqslant)$ *for each* $u \in V^+$.

Then for each $n \in \mathbb{N}$, distinct variable names $x_1, ..., x_n$, and a term $t(x_1, ..., x_n) \in T_{x_1,...,x_n}(V; p_1, ..., p_k; f_1, ..., f_l)$ we have

$[t] \in FnI^n_{CC}(V, A, \leqslant)$.

Proof. The proof can be straightforwardly done by induction on the structure of the term $t(x_1, ..., x_n)$ of the algebraic structure of nominative data $NDAS_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l)$ (note that the base of the induction follows from the assumptions). $\square$

Lemma 2. *Let $\leqslant$ be a transitive binary relation on $NDVC(V, A)$. Assume that:*

$f_1, f_2, ..., f_l, \Rightarrow u, u \Rightarrow_a, \nabla^u_a \in FnI_{CC}(V, A, \leqslant)$ *for each* $u \in V^+$.

$p_1, p_2, ..., p_k, IsEmpty \in PrM_{CC}(V, A, \leqslant), u!,$ *for each* $u \in V^+$.

Then for each $n \in \mathbb{N}$, distinct variable names $x_1, ..., x_n$, a term $t(x_1, ..., x_n) \in T_{x_1,...,x_n}(V; p_1, ..., p_k; f_1, ..., f_l)$, and a formula $\varphi(x_1, ..., x_n) \in \Phi_{x_1,...,x_n}(V; p_1, ..., p_k; f_1, ..., f_l)$ we have

$[t] \in FnI^n_{CC}(V, A, \leqslant)$ *and* $[\varphi] \in PrM^n_{CC}(V, A, \leqslant)$.

Proof. The proof can be straightforwardly done by induction on the structure of the term $t(x_1, ..., x_n)$ and the formula $\varphi(x_1, ..., x_n)$ of the algebraic structure of nominative data $NDAS_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l)$ (note that the base of the induction follows from the assumptions). $\square$

Lemma 3. *Let $\leqslant$ be a preorder $NDVC(V, A)$. Assume that:*

$f_1, f_2, ..., f_l, \Rightarrow u, u \Rightarrow_a, \nabla^u_a, u! \in FnM_{CC}(V, A, \leqslant)$ *for each* $u \in V^+$ *and* $p_1, p_2, ..., p_k, IsEmpty \in PrM_{CC}(V, A, \leqslant)$.

Then for each $n \in \mathbb{N}$, distinct variable names $x_1, ..., x_n$, and a term $t(x_1, ..., x_n) \in T_{x_1,...,x_n}(V; p_1, ..., p_k; f_1, ..., f_l)$, and a formula $\varphi(x_1, ..., x_n) \in \Phi_{x_1,...,x_n}(V; p_1, ..., p_k; f_1, ..., f_l)$ we have

$[t] \in FnM^n_{CC}(V, A, \leqslant)$ *and* $[\varphi] \in PrM^n_{CC}(V, A, \leqslant)$.

Proof. The proof can be straightforwardly done by induction on the structure of the term $t(x_1, ..., x_n)$ and the formula $\varphi(x_1, ..., x_n)$ of the algebraic structure of nominative data $NDAS_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l)$. □

Theorem 2. *(1) If $\leqslant$ is a transitive relation on $NDVC(V, A)$ and the conditions of Lemma 1 hold, then*

$FnEds_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l) \subseteq FnI_{CC}(V, A, \leqslant)$.

(2) If $\leqslant$ is a transitive relation on $NDVC(V, A)$ and the conditions of Lemma 2 hold, then

$PrEds_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l) \subseteq PrM_{CC}(V, A, \leqslant)$ *and*

$FnEds_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l) \subseteq FnI_{CC}(V, A, \leqslant)$.

(3) If $\leqslant$ is a preorder on $NDVC(V, A)$ and the conditions of Lemma 3 hold, then

$PrEds_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l) \subseteq PrM_{CC}(V, A, \leqslant)$ *and*

$FnEds_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l) \subseteq FnM_{CC}(V, A, \leqslant)$.

Proof. (1) Let us show that $FnEds_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l) \subseteq FnI_{CC}(V, A, \leqslant)$. Let $f \in FnEds_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l)$, $d \in NDVC(V, A)$, and $f(d)\downarrow$. Then using notations of Definition 13 we can say that there exists i such that $[\varphi_i](d, d_1, ..., d_n) \downarrow= T$ and $[t_i](d, d_1, ..., d_n) \downarrow= f(d)$. By Lemma 2, we have $[t_i] \in FnI^{n+1}_{CC}(V, A, \leqslant)$, so $d \leqslant [t_i](d, d_1, ..., d_n) = f(d)$. Since d is arbitrary, we have $f \in FnI_{CC}(V, A, \leqslant)$.

(2) Let us show that

$PrEds_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l) \subseteq PrM_{CC}(V, A, \leqslant)$.

Let $p \in PrEds_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l)$, $d', d'' \in NDVC(V, A)$ and $d' \leqslant d''$. Assume that $p(d')\downarrow$. Then using notations of Definition 14 we can say that there exists i such that $p(d') = b_i$ and $[\varphi_i](d', d_1, ..., d_n) \downarrow= T$. By Lemma 2, $[\varphi_i] \in PrM^{n+1}_{CC}(V, A, \leqslant)$, so $[\varphi_i](d'', d_1, ..., d_n) \downarrow= T$. Then $p(d'') \downarrow= b_i = p(d')$. Since d is arbitrary, we have $p \in PrM_{CC}(V, A, \leqslant)$.

That $FnEds_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l) \subseteq FnI_{CC}(V, A, \leqslant)$ can be shown similarly to the case (1) above.

(3) Let us show that

$PrEds_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l) \subseteq PrM_{CC}(V, A, \leqslant)$.

Let $p \in PrEds_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l)$, $d', d'' \in NDVC(V, A)$ and $d' \leqslant d''$. Assume that $p(d') \downarrow$. Then using notations of Definition 14 we can say that there exists i such that $p(d') = b_i$ and $[\varphi_i](d', d_1, ..., d_n) \downarrow = T$. By Lemma 3, $[\varphi_i] \in PrM^{n+1}_{CC}(V, A, \leqslant)$, so $[\varphi_i](d'', d_1, ..., d_n) \downarrow = T$. Then $p(d'') \downarrow = b_i = p(d')$. Since d is arbitrary, we have $p \in PrM_{CC}(V, A, \leqslant)$.

Let us show that

$FnEds_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l) \subseteq FnM_{CC}(V, A, \leqslant)$.

Let $f \in FnEds_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l)$, $d, d' \in NDVC(V, A)$, $d \leqslant d'$, and $f(d) \downarrow$. Then using notations of Definition 13 we can say that there exists i such that $[\varphi_i](d, d_1, ..., d_n) \downarrow = T$ and $[t_i](d, d_1, ..., d_n) \downarrow = f(d)$. By Lemma 3, $[t_i] \in FnM^{n+1}_{CC}(V, A, \leqslant)$ and $[\varphi_i] \in PrM^{n+1}_{CC}(V, A, \leqslant)$, so $[t_i](d', d_1, ..., d_n) \downarrow$ and $f(d) = [t_i](d, d_1, ..., d_n) \leqslant [t_i](d', d_1, ..., d_n)$. Moreover, $[\varphi_i](d', d_1, ..., d_n) \downarrow = T$, so $f(d') \downarrow = [t_i](d', d_1, ..., d_n)$ and $f(d) \leqslant f(d')$. Since d is arbitrary, we have $f \in FnM_{CC}(V, A, \leqslant)$.

□

Corollary 1. *Under the conditions of Lemma 1 or Lemma 2, all unary functions (programs) expressible in $NGA^a_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l)$ belong to $FnI_{CC}(V, A, \leqslant)$.*

Proof. Follows immediately from Theorem 1 and Theorem 2. □

Corollary 2. *Under the conditions of Lemma 3, all unary functions (programs) expressible in $NGA^a_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l)$ belong to $FnM_{CC}(V, A, \leqslant)$.*

Proof. Follows immediately from Theorem 1 and Theorem 2. □

Corollary 1 implies that in order to show that a program expressible in $NGA^a_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l)$ has the property $(d, f(d)) \in \leqslant$ which expresses the fact that the input and output of f belong to a transitive relation $\leqslant$, it is sufficient to check several properties of basic operations on nominative data with respect to $\leqslant$ which are formulated in Lemma 1. It is not necessary to prove preservation of this property by the compositions of $NGA^a_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l)$, since this preservation follows automatically from eds definability of all programs expressible in $NGA^a_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l)$.

It is easy to see that the results similar to corollaries from Theorem 2 hold not only for eANGAA, but for a reduct of ANGAA [18], i.e. an algebra with narrower carriers and/or sets of operations (in particular, constants) and predicates:

1) if all function constants of a reduct of eANGAA are $\leqslant$-increasing, then all unary functions (programs) expressible in this reduct are $\leqslant$-increasing.

2) if in a reduct of eANGAA all function constants are $\leqslant$-monotone and all predicate constants are $\leqslant$-equitone, then all unary functions (programs) expressible in this reduct are $\leqslant$-monotone.

Using this observation, e.g., we can show partial correctness of the GCD program mentioned in the introduction by considering a reduct of eANGAA of functions and predicates over $NDVC(\{x, y, a, b\}, \mathbb{Z})$ which has as function constants the functions which perform assignment operations which appear in this program: $f_1(d) = d\nabla_a[a \mapsto d(x)]$, $f_2(d) = d\nabla_a[b \mapsto d(y)]$, $f_3(d) = d\nabla_a[a \mapsto d(a) - d(b)]$, $f_4(d) = d\nabla_a[b \mapsto d(b) - d(a)]$ and showing that they are $\leqslant$-increasing.

Corollary 2 implies that in order to show that a program expressible in $NGA^a_{CC}(V, A; p_1, ..., p_k; f_1, ..., f_l)$ is monotone with respect to some preorder on data, it is sufficient to check several properties of basic operations on nominative data with respect to $\leqslant$ which are formulated in Lemma 3.

An example of application of the obtained results is given below. In [11] the authors considered a special property of programs which is called *nominative stability* [11], [8], [9], [10]. This property is a formalization of the idea of stability of program semantics when the data structures used in the program are changed to equivalent in the sense of information content and supported operations.

It can be illustrated by the following feature of the Pascal programming language: the two-dimensional array definitions `var A: array [1..n, 1..m] of real` and `var A:array [1..n] of array [1..m] of real` are equivalent and both the `A[i,j]` and `A[i][j]` syntax can be used to access the array elements regardless of the form of its definition (it should be noted that the languages like C++ and Java do not have this feature). This implies that one can safely swap two-dimensional array definitions in a program without changing the rest of the text of the program while preserving program semantics.

Nominative stability is defined using the *nominative equivalence* relation on nominative data of the type TND_{CC}. This relation is a formalization of the idea that data are equivalent, if they have essentially the same information content, but may have different hierarchical naming structure. For example, the following data are nominatively equivalent: $[v_1 \mapsto [v_2 \mapsto [v_3 \mapsto 1]]]$ and $[v_1 v_2 v_3 \mapsto 1]$, as they differ only in the naming hierarchy, but contain the same basic names and values. A function on nominative data is nominative stable, if on nominative equivalent data it gives nominative equivalent results.

Formally,

Definition 15. *1) Data $d_1 \in NDVC(V, A)$ is nominatively included in $d_2 \in NDVC(V, A)$, if either $d_1, d_2 \in A$ and $d_1 = d_2$, or $d_1, d_2 \notin A$ and for each terminal path $(v_1, v_2, ..., v_n)$ in d_1 there is a terminal path $(v'_1, v'_2, ..., v'_m)$ in d_2 such that $v_1 v_2 ... v_n = v'_1 v'_2 ... v'_m$ and the values of $(v_1, v_2, ..., v_n)$ in d_1 and $(v'_1, v'_2, ..., v'_m)$ in d_2 coincide.*

2) Data d_1, d_2 are nominative equivalent ($d_1 \approx d_2$), if d_1 is nominatively included in d_2 and d_2 is nominatively included in d_1.

3) The elements of $FnM_{CC}(V, A, \approx)$ are called nominative stable functions (programs).

Using Corollary 2 formulated above we can easily show that all programs expressible in $NGA^a_{CC}(V, A)$ (i.e. ANGAA without additional predicates and functions) are nominative stable.

In [11] it was shown that $\approx$ is an equivalence on $NDVC(V, A)$ (and thus is a preorder) and the functions $\Rightarrow u$, $u \Rightarrow_a$, ∇^u_a are nominative stable. It is trivial to check by the definition that $u!$ is $\approx$-equitone binary predicate on $NDVC(V, A)$ and $IsEmpty$ is $\approx$-monotone (i.e. nominative stable). Then by Corollary 2, all functions expressible in $NGA^a_{CC}(V, A)$ are nominative stable.

7 Conclusions

We have investigated methods of proving properties of programs on hierarchical nominative data on the basis of the composition-nominative approach. The proofs of properties of programs depend on proofs of properties of compositions and basic operations on data. The complexity of such proofs can be lowered, if one is able to reduce the proofs of properties of various compositions to proofs of properties of operations on data. We have proposed a way of achieving such a reduction by representing compositions using effective definitional schemes of H. Friedman. The achieved reduction permits us to consider proofs in data algebras (which are simpler to derive, automate, etc.) instead of proofs in program algebras. Using this approach we have demonstrated that the properties of programs related to structural transformations of data can be reduced to the data level. The obtained results can be used in software development and verification.

References

[1] T. Hoare, “The verifying compiler: A grand challenge for computing research,” *Journal of the ACM*, vol. 50, no. 1, pp 63–69, 2003.

[2] R. W. Floyd, “Assigning meanings to programs,” in *Proceedings of Symposium on Applied Mathematics*, vol. 19, J.T. Schwartz, Ed. A.M.S., 1967, pp. 19–32.

[3] C. A. R. Hoare, “An axiomatic basis for computer programming,” *Communications of the ACM*, vol. 12, no. 10, pp. 576–580, 1969.

[4] B. Meyer, "Design by Contract," Interactive Software Engineering Inc., Technical Report TR-EI-12/CO, 1986.

[5] N. S. Nikitchenko, "A composition-nominative approach to program semantics," Technical University of Denmark, Technical report IT-TR 1998-020, 1998.

[6] M. S. Nikitchenko and S. S. Shkilniak, *Mathematical logic and theory of algorithms*, Publishing house of Taras Shevchenko National University of Kyiv, 2008, 528 p. ISBN: 966-439-007-0. (in Ukrainian)

[7] A. Kryvolap, M. Nikitchenko and W. Schreiner, "Extending Floyd-Hoare logic for partial pre- and postconditions," in *Information and Communication Technologies in Education, Research, and Industrial Applications*, vol. 412 of Communications in Computer and Information Science, Springer International Publishing, 2013, pp. 355–378.

[8] M. Nikitchenko and Ie. Ivanov, "Programming with nominative data," in *Proceedings of CSE'2010 International Scientific Conference on Computer Science and Engineering*, September 20-22, 2010, Kosice, Slovakia, 2010, pp. 30–39.

[9] M. S. Nikitchenko and Ie. Ivanov, "Stability and monotonicity of programs with respect to structure transformations of data," *Problems in programming*, no. 2–3, pp. 58–67, 2010. (in Ukrainian)

[10] M. S. Nikitchenko and Ie. Ivanov, "Composition-nominative languages of programs with associative denaming," *Bulletin of Lviv University, Ser. Appl. Math. Inform.*, no. 16, pp. 124–139, 2010. (in Ukrainian)

[11] V. G. Skobelev, M. Nikitchenko and Ie. Ivanov, "On algebraic properties of nominative data and functions," in *Information and Communication Technologies in Education, Research, and Industrial Applications*, vol. 469, Communications in Computer and In-

formation Science, Springer International Publishing, 2014, pp. 117–138.

[12] M. S. Nikitchenko and V. G. Tymofieiev, “Satisfiability in composition-nominative logics,” *Central European Journal of Computer Science*, vol. 2, no. 3, pp. 194–213, 2012.

[13] V. G. Skobelev, Ie. Ivanov and M. Nikitchenko, “Set-theoretic analysis of nominative data,” *Computer Science Journal of Moldova*, vol. 23, no. 3(69), pp. 270–288, 2015.

[14] V. M. Glushkov, “Automata theory and formal transformations of microprograms,” *Cybernetics*, no. 5, pp. 1–10, 1965. (in Russian)

[15] H. Friedman, “Algorithmic procedures, generalized Turing algorithms, and elementary recursion theory,” in *LOGIC COLLOQUIUM ’69*, Studies in Logic and the Foundations of Mathematics, Elsevier, vol. 61, pp. 361–389, 1971.

[16] H. Friedman and R. Mansfield, “Algorithmic Procedures,” *Trans. Amer. Math. Soc.*, vol. 332, pp. 297–312, 1992.

[17] H. R. Nielson and F. Nielson, *Semantics with applications: a formal introduction*, Wiley Professional Computing, John Wiley & Sons, Inc., New York, 1992, 252 p., ISBN: 0-471-92980-8.

[18] P. Burmeister, “Lecture notes on universal algebra. Many-sorted partial algebras,” 2002. [Online]. Available: `http://www.mathematik.tu-darmstadt.de/Math-Net/Lehrveranstaltungen/Lehrmaterial/SS2002/AllgemeineAlgebra/download/LNPartAlg.pdf`

Ievgen Ivanov, Mykola Nikitchenko,
Volodymyr G. Skobelev

Received October 6, 2016

Ievgen Ivanov
Taras Shevchenko National University of Kyiv
64/13 Volodymyrska Street, 01601 Kyiv, Ukraine
Phone: +380442590519
E–mail: `ivanov.eugen@gmail.com`

Mykola Nikitchenko
Taras Shevchenko National University of Kyiv
64/13 Volodymyrska Street, 01601 Kyiv, Ukraine
Phone: +380442590519
E–mail: `nikitchenko@unicyb.kiev.ua`

Volodymyr G. Skobelev
V.M. Glushkov Institute of Cybernetics of NAS of Ukraine
40 Glushkova ave., Kyiv, Ukraine, 03187
Phone: +380634318605
E–mail: `skobelevvg@mail.ru`