On Algebraic Properties of Nominative Data and Functions

Volodymyr G. Skobelev[1], Mykola Nikitchenko[2], and Ievgen Ivanov[2](✉)

[1] Institute of Applied Mathematics and Mechanics of NAS of Ukraine, Donetsk, Ukraine
skbv@iamm.ac.donetsk.ua

[2] Taras Shevchenko National University of Kyiv, Kyiv, Ukraine
nikitchenko@unicyb.kiev.ua, ivanov.eugen@gmail.com

Abstract. In the chapter basic properties of nominative data and functions over nominative data (nominative functions) are investigated from the perspective of abstract algebra. A set of all nominative data over arbitrary fixed sets of names and values together with basic operations which include naming, denaming, and overlapping is considered as an algebraic structure and its main properties are studied. Nominative data with complex names satisfy the principle of associative naming and processing. For such data a natural equivalence relation is introduced. Properties of nominative functions (mathematical models of programs over nominative data) and predicates are studied. A notion of nominative stability of nominative functions and predicates is considered. A two-sorted algebra of nominative functions and predicates which generalizes Glushkov algorithmic algebras is introduced and it is proved that the set of nominative stable functions and the set of nominative stable predicates constitute its sub-algebra. The obtained results form a mathematical basis for nominative program logic construction.

Keywords: Glushkos algorithmic algebra · Program algebra · Nominative data · Nominative set · Named set · Nominative function

1 Introduction

To increase software system reliability a number of mathematically based approaches for software (and hardware) development can be used. These approaches are usually referred to as formal methods. Such methods are grounded on different branches of mathematical logic, set theory, automata theory, universal algebra, formal language theory, and other fundamental areas of mathematics. The central idea of formal methods can be described as follows: first, a formal specification of a software system (abstract system model) is described, then this

This work was supported in part by the research project No. 11BF015-02 "Formal specifications and methods of development of reliable software systems", Taras Shevchenko National University of Kyiv, Ukraine.

V. Ermolayev et al. (Eds.): ICTERI 2014, CCIS 469, pp. 117–138, 2014.
DOI: 10.1007/978-3-319-13206-8_6

specification is successively transformed to a concrete specification (system realization), and, at last, correctness of transformations can be proved. The majority of formal methods are based on algebraic approach. This approach has the following two characteristics: (1) the formalism of many-sorted algebras is used to model software systems; (2) special logics based on such many-sorted algebras are used to reason about systems and their transformations. In the literature various kinds of such algebras and logics are described (for instance, see [1]).

In this chapter we use composition-nominative approach [2] to construct algebras of nominative data and function, considered as formal models of program, and investigate properties of these algebras. This approach is grounded on several principles [2–4] including the *Development principle* (from abstract to concrete), *Principle of integrity of intensional and extensional aspects*, *Principle of priority of semantics over syntax*, *Compositionality principle*, *Nominativity principle*.

The latter Nominativity principle states that nominative data [2] are adequate mathematical models of various forms of data that are processed and stored in computing systems. There are several types of nominative data [5,6], and all of them are based on naming relations that associate names and values. The simplest type of nominative data is a nominative set (or named set) [2,5] which is defined as a partial function from an arbitrary set of names to an arbitrary class values. Other types of nominative data represent hierarchical data organizations with simple or complex names.

On the abstract level a computing system is modeled as a partial function that maps nominative data (input data) to nominative data (output data). Such functions are called binominative. More generally, one can consider functions which map nominative data to arbitrary values. Nominative functions can be composed in various ways, e.g. by sequential composition, branching, etc. Operations that construct composed systems from constituents are called compositions.

A set of compositions available to a system developer together with a set of functions obtained from some chosen set of basic functions by applications of compositions form a certain algebraic system (program algebra) which is considered as a semantic model of a computing system development language. A syntax of this development language follows naturally from this semantic model: programs are represented as terms of the described algebra.

The relation of the above mentioned notions to semantics of programming languages can be illustrated on a simple educational programming language WHILE described in [7]. This is an imperative language in which programs are composed from statements that involve boolean and arithmetic expressions. A program state is an assignment of values to variable names. It can be modeled as a nominative set (which is defined on assigned variables and is undefined on unassigned variables). Semantics of a statement can be represented as a (partial) binominative function (a mapping from states to states) and semantics of a boolean or arithmetic expression can be represented as a function on nominative data which takes boolean or, respectively, integer values. Semantics of statements are composed to obtain semantics of a program.

The described approach allows one to investigate semantic properties of programs and develop methods of synthesis of systems with desired properties. In particular, in the works on the composition-nominative approach [2–4] various logical systems for reasoning about properties of computing systems (functions) and solving verification problems have been proposed. Some of them [4] are based on the classical Floyd-Hoare logic [8,9], but extend it to handle a larger class of pre- and post-conditions that includes partial predicates.

Although applications are important for the composition-nominative approach, it is largely a mathematical framework, so it gives a rise to various theoretical questions.

One such question is definition and investigation of algebras of programs over nominative data. This question is the topic of this chapter. A set of functions and a set of predicates over nominative data which can be obtained from basic functions and predicates using compositions such as branching, cycle, etc. [2,6,10] form a two-sorted algebraic structure [11] which generalizes Glushkov algorithmic algebras [12]. However, its main difference and advantage is that its functions are defined over nominative data which give a natural representation of data structures commonly used in programming. We will define and investigate some properties of this algebra in this chapter.

The chapter is organized in the following way: in Sect. 2 we will introduce the necessary notation; in Sect. 3 we will give an overview of the composition-nominative approach; in Sect. 4 we will introduce the formal definition and classification of nominative data; in Sect. 5 we will describe representation of various concrete data structures by nominative data; in Sect. 6 we will introduce the main operations over nominative data; in Sect. 7 we will introduce algebras of nominative data; in Sect. 8 we will consider the notion of nominative equivalence and its relation to algebras of nominative data; in Sect. 9 we will introduce the main compositions of functions and predicates over nominative data and introduce a generalization of Glushkov algorithmic algebras which we call Associative Nominative Glushkov Algorithmic Algebras; in Sect. 10 we introduce the notion of nominative stability of nominative predicates and functions and prove that the sets of nominative stable functions and predicates form a sub-algebra of an Associative Nominative Glushkov Algorithmic Algebra and discuss practical implications of this result.

2 Notation

We will use the following notation:

- 2^A is the power set of a set A;
- $|A|$ means the cardinality of a set A;
- $A \tilde{\to} B$ is the class of partial functions from A to B;
- $f(x)\downarrow$ (or $f(x)\uparrow$) means that a partial function f is defined (or respectively, undefined) on an argument x. The notation $f(x)\downarrow Ry$, where R is a binary relation e.g. equality ($=$), inclusion ($\in$), etc. means that $f(x)$ is defined and the relation $f(x)Ry$ holds;

– Dom f denotes the domain of a function, i.e. if $f : A \tilde{\rightarrow} B$, then $Dom f = A$;
– $dom(f)$ denotes the set of arguments on which f is defined (domain of definedness), i.e. $dom(f) = \{x \mid f(x) \downarrow\}$;
– $im(f) = \{f(x) \mid x \in dom(f)\}$ is the image of a function f;
– $f|_A$ is the restriction of a function f to a set A;
– $f^{-1}(A) = \{x \mid f(x) \downarrow \in A\}$ is the preimage of a set A under a function f;
– $f(x) \cong g(y)$ means the strong equality: if at least one of the values $f(x)$ and $g(y)$ is defined, then the another one is defined and they both are equal.

3 Composition-Nominative Approach to Program Formalization

In order to discuss role of data in programming we should construct adequate models of programs. To tackle this problem we use composition-nominative approach to program formalization [4]. This approach aims to construct a hierarchy of program models of various abstraction levels and generality and is based on the following main principles:

– *Development principle* (from abstract to concrete): program notions should be introduced as a process of their development that starts from abstract understanding and proceeds to more concrete considerations.
– *Principle of integrity of intensional and extensional aspects*: program notions should be presented in the integrity of their intensional and extensional aspects. The intensional aspects in this integrity play a leading role.
– *Principle of priority of semantics over syntax*: program semantic and syntactical aspects should be first studied separately, then in their integrity in which semantic aspects prevail over syntactical ones.
– *Compositionality principle*: programs can be constructed from simpler programs (functions) with the help of special operations, called compositions, which form a kernel of program semantics structures.
– *Nominativity principle*: nominative (naming) relations are basic ones in constructing data.

According to the composition-nominative approach program models are specified as *composition-nominative systems* (CNS). These systems are composed of simpler systems: composition, description, and denotation systems. Composition system defines semantic aspects of programs, description system defines program descriptions (syntactical aspects), and denotation system specifies meanings of descriptions.

We consider semantics of programs as partial functions over class of data processed by programs; compositions are n-ary operations over functions. Thus, a composition system can be specified as two algebras: data algebra and function algebra. Function algebra is the main semantic notion in program formalization. Terms of this algebra define syntax of programs (descriptive system), and ordinary procedure of term interpretation gives a denotation system.

CNS can be used to construct formal models of various programming, specification, and database languages [4,5]. The program models represented by CNS are mathematically simple, but specify program semantics rather adequately. They are classified in accordance with the levels of abstraction of their parameters: data, functions, and compositions. Usually abstraction levels of program models are induced by abstraction levels of data.

Data are considered on three levels: abstract, boolean, and nominative. On the abstract level data are treated as "black boxes", thus no information can be extracted. On the boolean level to abstract data new data considered as "white boxes" are added. Usually, these are logical values T (true) and F (false) from the set *Bool*. On the nominative level data are considered as "grey boxes", constructed of "black boxes" (values) and "white boxes" (names) with the help of naming relations. The last level is the most interesting for programming. Data of this level are called *nominative data*.

Concretizations of nominative data can represent various data structures, such as records, arrays, lists, relations, etc. Thus, we can formulate the following *data representativity principle*: program data can be represented as concretizations of nominative data. To support this principle later in the chapter we will give representations of various data structures using nominative data.

We start with rigorous definitions of various classes of nominative data.

4 Classification of Nominative Data

Nominative data are built over classes of names V and atoms (basic values) A with the help of naming relations. Thus in the first approximation, a nominative data d is either an atom from A, or has the form $[v_1 \mapsto d_1, ..., v_n \mapsto d_n]$, where $v_1, ..., v_n$ are distinct names from V and $d_1, d_2, ...d_n$ are atoms or possibly other nominative data.

To define nominative data let us denote by $V \xrightarrow{n} B$ the class of all partial functions from a set of names V to a set of values B which have a finite graph.

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).

The described parameters give 4 types of nominative data. There can be different ways to define the notion of a complex name. We will follow the principle of development from abstract to concrete and consider the simplest case of name construction and processing. This means that we will consider complex names as *sequences* of simple names that satisfy the *associativity* property. In other words, names are constructed with the help of concatenation operation which is associative by definition. This allows us to formulate *the principle of associative construction and processing of complex names*: complex names are constructed from simple names using concatenation and data with complex names have to be processed by operations that take into account associativity of names. Besides, we will require that data with complex names satisfy *the principle of unambiguous associative naming* which means that one complex name should have at most one corresponding value in any given data.

The formal definitions of nominative data of different types and the corresponding examples are given below.

- $D_0 = V \xrightarrow{n} A$. This is the simplest type of nominative data. For example,

$$[u \mapsto 1, v \mapsto 2].$$

  We will say that members of D_0 are data of the type TND_{SS} (data with simple names and simple values).
- $D_1 = ND(V, A)$, where $ND(V, A)$ is defined as

$$ND(V, A) = \bigcup\nolimits_{k \geq 0} ND_k(V, A),$$

  where

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

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

  Note that here we denote by $\emptyset$ the empty nominative data (this notation is also used for the empty set). For the empty nominative data we will also use the notation [].
  Data from this class are hierarchically constructed, for example,

$$[u \mapsto 1, v \mapsto [w \mapsto 2]].$$

  Such data can be represented by oriented trees (of varying arity) with arcs labelled by names and with leafs labelled by atoms. A *path* is a nonempty finite sequence $(v_1, v_2, ..., v_k)$ of names $v_1, ..., v_k \in V$. For a given data d a *value of a path* $(v_1, v_2, ..., v_k)$ in d is defined by the following 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 called the *rank* of a data d.
  We will say that members of D_1 are data of the type TND_{SC} (data with simple names and complex values).
- $D_2 = NDVS(V, A)$, 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 (*the principle of unambiguous associative naming*).

For example

$$[uv \mapsto 1, uw \mapsto 2, w \mapsto 3].$$

Data of this class have *complex names* i.e. names that are strings in alphabet V.
We will say that members of D_2 are data of the type TND_{CS} (data with complex names and simple values).

- $D_3 = NDVC(V, A)$, 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 $u_1 u_2 ... u_k$ and $v_1 v_2 ... v_l$ are incomparable is sense of the prefix relation (*principle of unambiguous associative naming*). Such data are also called *complex-named data* [6]. For example,

$$[uv \mapsto 1, w \mapsto [uw \mapsto \emptyset]].$$

  These data are hierarchically constructed data with complex names and unambiguous naming.
  We will say that members of D_3 are data of the type TND_{CC} (data with complex names and complex values).

5 Examples of Representations of Data Structures by Nominative Data

In this section we present some concrete examples to support data representativity principle: program data can be adequately represented as concretizations of nominative data.

Below we describe representations of various commonly used data structures by nominative data. Representations are chosen in such way that the basic operations over data structures correspond to simple combinations of basic operations over nominative data.

- Array $(a_1, a_2, ..., a_n)$ of elements of type T_0.
  Nominative data type: TND_{SS}, $V = \{1, 2, \ldots\}$, $A = T_0$.
  Representation:

$$[1 \mapsto a_1, 2 \mapsto a_2, \ldots, n \mapsto a_n].$$

- Two-dimensional array $\begin{pmatrix} a_{1,1} & a_{1,2} & \cdots & a_{1,m} \\ \cdots & \cdots & \cdots & \cdots\cdot \\ a_{n,1} & a_{n,2} & \cdots & a_{n,m} \end{pmatrix}$ of elements of type T_0.
  Nominative data type: TND_{CS} (complex names), $V = \{1, 2, \ldots\}$, $A = T_0$.
  Representation:

$$[i.j \mapsto a_{i,j} \mid i = 1, \ldots, n,\ j = 1, \ldots, m]$$

  (in this notation the dot "." separates symbols in a complex name).

– Jagged array $((a_{i,j})_{j=1}^{m})_{i=1}^{n}$ of elements of type T_0.
Nominative data type: TND_{SC}, $V = \{1, 2, \ldots\}$, $A = T_0$.
Representation:

$$[i \mapsto [j \mapsto a_{i,j} \mid j = 1, \ldots, m] \mid i = 1, \ldots, n].$$

– Associative array $(a_k)_{k \in K}$ of elements of type T_0, where K is a set of keys.
Nominative data type: TND_{SS}, $V = K$, $A = T_0$.
Representation:

$$[k \mapsto a_k \mid k \in K].$$

– Table of the form (with Key being a key attribute):

Key	Attr1	Attr2
key_1	val_{11}	val_{21}
key_2	val_{12}	val_{22}

Nominative data type: TND_{SC}, $V = \{Attr_1, Attr_2\}$.
Representation:

$$\begin{aligned}&[key_1 \mapsto [Attr_1 \mapsto val_{11}, Attr_2 \mapsto val_{12}],\\ &key_2 \mapsto [Attr_1 \mapsto val_{21}, Attr_2 \mapsto val_{22}]].\end{aligned}$$

– A list of elements $e_1, e_2, \ldots e_n$.
Nominative data type: TND_{SC}, $V = \{data, next\}$.
Representation:

$$[data \mapsto e_1, next \mapsto [data \mapsto e_2, next \mapsto [\ldots data \mapsto e_n, next \mapsto \emptyset \ldots]]].$$

– A circular list of elements $e_1, e_2, ...e_n$.
Nominative data type: TND_{SC}, $V = \{head, memory, data, next\} \cup \{loc_1, loc_2, loc_3, \ldots loc_n\}$.
Representation:

$$\begin{aligned}&[head \mapsto loc_1, memory \mapsto [\\ &\qquad loc_1 \mapsto [data \mapsto e_1, next \mapsto loc_2],\\ &\qquad loc_2 \mapsto [data \mapsto e_2, next \mapsto loc_3],\\ &\qquad \ldots\\ &\qquad loc_n \mapsto [data \mapsto e_n, next \mapsto loc_1]]].\end{aligned}$$

– A binary tree with nodes labelled by elements $e_1, e_2, e_3, ...$
Nominative data type: TND_{SC}, $V = \{data, left, right\}$.
Representation:

$$\begin{aligned}&[data \mapsto e_1,\\ &left \mapsto [data \mapsto e_2, left \mapsto [\ldots], right \mapsto [\ldots]],\\ &right \mapsto [data \mapsto e_3, left \mapsto [\ldots], right \mapsto [\ldots]\\ &]].\end{aligned}$$

– Data types based on free algebras (simple forms of inductive data types [13]). Such data are widely used in (partly) functional programming languages (e.g. ML, Haskell). They are suitable for representing list- and tree-like data. Such types can be interpreted as carrier sets of (many-sorted) free algebras or as sets of terms.

  Let S be a finite set of elements called *sorts*. Let Σ be finite *signature* $\{c_1, c_2, ..., c_n\}$, where c_i are (names of) *constructors*. Suppose that each constructor c_i has (unique) associated *type* of the form $s_1 \times \ldots \times s_m \to s$, where $s_i, s \in S$, $m \geq 0$. Let D be a free algebra with the signature Σ which is freely generated by some set A. Then the members of the carrier set of D can be considered as elements of an inductive data type. Data of such type can be processed by means of application of constructors and pattern matching (for data decomposition).

  Then one can define a mapping τ which gives a representation of data from D by nominative data of the type TND_{SC} over the classes of names $V = \{constructor, 1, 2, 3, \ldots\}$ and the basic values A:

  - $\tau(x) = x$, if $x \in A$.
  - $\tau(x) = [constructor \mapsto c, 1 \mapsto \tau(x_1), 2 \mapsto \tau(x_2), ..., n \mapsto \tau(x_n)]$, if x has a form $c(x_1, x_2, \ldots, x_n)$, where $c \in \Sigma$.

  Applications of constructors and pattern matching on inductive data types correspond to naming and denaming (and name equality checking) on nominative data.

6 Operations over Nominative Data

The main operations over nominative data are the operations of *denaming* (taking the value of a name), *naming* (assigning a new value to a name), and *overlapping*.

In this section we will define these operations for data of the types TND_{CS}, TND_{SC}, and TND_{CC}. Analogous operations can be rather straightforwardly defined on data of the type TND_{SS}, but we will not consider them in this chapter.

We will use the same symbols for denoting denaming, naming, and overlapping for each type of data TND_{CS}, TND_{SC}, and TND_{CC}. The interpretation of the operation symbols should be clear from the context.

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

Definition 1 (Denaming).

(1) For nominative data of the type TND_{SC}, (associative) denaming is an unary operation $v \Rightarrow_a$ with a parameter $v \in V^+$ defined by induction by the length of v (denoted as $|v|$) as follows:
  - *if $|v| = 1$, then $v \Rightarrow_a (d) \cong d(v)$;*
  - *if $|v| = n > 1$, then $v \Rightarrow_a (d) \cong v_1 \Rightarrow_a (x \Rightarrow_a (d))$, where $v = xv_1$, $x \in V$, $v_1 \in V^{n-1}$.*

(2) For nominative data of the types TND_{CS} and TND_{CC}, (associative) denaming is an unary operation $v \Rightarrow_a$ with a parameter $v \in V^+$ defined by induction by the length of v:
– *if $|v| = 1$, then*

$$v \Rightarrow_a (d) \cong \begin{cases} d(v), & \textit{if } d(v) \downarrow; \\ d/v, & \textit{if } d(v) \uparrow \textit{ and } d/v \neq \emptyset; \\ \textit{undefined}, & \textit{if } d(v) \uparrow \textit{ and } d/v = \emptyset, \end{cases}$$

where

$$d/u = [v_1 \mapsto d(v) \mid d(v) \downarrow, v = uv_1, v_1 \in V^+]$$

(division of a data by a name);
– *if $|v| = n > 1$, then*

$$v \Rightarrow_a (d) \cong v_1 \Rightarrow_a (x \Rightarrow_a (d)),$$

where $v = xv_1$, $x \in V$, $v_1 \in V^+$, $|v_1| = n - 1$.

The following examples illustrate this definition:

$$u \Rightarrow_a ([u \mapsto 1, v \mapsto 2]) = 1;$$

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

The name of this operation originates from the following property (*associativity*) [6]:

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

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

Definition 2 (Naming).

(1) For nominative data of the type TND_{SC}, naming is an unary operation $\Rightarrow v$ with a parameter $v \in V^+$ defined inductively by the length of v as follows:
$\Rightarrow v(d) = [v \mapsto d]$, if $v \in V$;
$\Rightarrow v(d) = [v_1 \mapsto (\Rightarrow v_2(d))]$, if $v = v_1 v_2$, $v_1 \in V$ and $v_2 \in V^+$.

(2) For nominative data of the type TND_{CS}, naming is an unary operation $\Rightarrow v$ with a parameter $v \in V^+$ defined as follows:
$\Rightarrow v(d) = [v \mapsto d]$, if $d \in A \cup \{\emptyset\}$;
$\Rightarrow v(d) = [vu \mapsto d(u) \mid u \in dom(d)]$, if $d \notin A \cup \{\emptyset\}$.

(3) For nominative data of the type TND_{CC}, naming is an unary operation $\Rightarrow v$ with a parameter $v \in V^+$ such that $\Rightarrow v(d) = [v \mapsto d]$.

Overlapping can be intuitively treated as an updating operation which updates values in the first argument with the values of the second argument taking into account their names.

For the types of nominative data with complex names and/or values 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 while the local one ∇_a^v (with a parameter name v) updates only one value with complex name v.

The global overlapping can be used for formalization of procedures calls and the local operation formalizes the assignment operator in programming languages. Intuitively, this operation joins two data and resolves name conflicts in favour of its *second* argument.

Definition 3 (Global overlapping).

(1) For nominative data of the type TND_{SC}, global overlapping is a partial binary operation ∇_a such that
$d_1 \nabla_a d_2 = d_2 \cup d_1|_{dom(d_1)\setminus dom(d_2)}$, if $d_1 \notin A$ and $d_2 \notin A$;
$d_1 \nabla_a d_2 \uparrow$, if $d_1 \in A$ or $d_2 \in A$.

(2) For nominative data of the type TND_{CS}, global overlapping is a binary operation ∇_a such that
$d_1 \nabla_a d_2 = d_2 \cup d_1|_{dom(d_1)\setminus(dom(d_2)V^*)}$;
$d_1 \nabla_a d_2 \uparrow$, if $d_1 \in A$ or $d_2 \in A$.
where $dom(d_2)V^*$ denotes the all words of the form uv, where $u \in dom(d_2)$ and $v \in V^*$ (i.e. an arbitrary, possibly empty word in the alphabet V).

(3) 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 less than or equal to k.
Induction base of the definition. If $d_1 \in NDVC_0(V, A)$, then

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

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

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

Then $d_1 \nabla_a d_2 = d$, where d is defined by its values on names $u \in V^+$ 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 $d_2/u = [v_1 \mapsto d_2(v) \mid d_2(v) \downarrow, v = uv_1, v_1 \in V^+]$ is the division of a data by a name;
(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 on the data of the type TND_{CC} has the following properties [6]:

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

Definition 4 (Local overlapping).

(1) For nominative data of the type TND_{CS} local overlapping is a binary operation ∇_a^v with a parameter $v \in V^+$ defined as follows:

$$d_1\nabla_a^v d_2 \cong d_1\nabla_a(\Rightarrow v(d_2)).$$

(2) For nominative data of the type TND_{SC} local overlapping is a binary operation ∇_a^v with a parameter $v \in V^+$ defined inductively by the length of v as follows:

- *if $v \in V$, then*
$$d_1\nabla_a^v d_2 \cong d_1\nabla_a[v \mapsto d_2];$$
- *if $v = v_1v_2$, where $v_1 \in V$ and $v_2 \in V^+$, and $d_1(v_1) \downarrow$ and $d_1(v_1) \notin A$, then*
$$d_1\nabla_a^v d_2 \cong d_1\nabla_a[v_1 \mapsto d_1(v_1)\nabla_a^{v_2} d_2];$$
- *if $v = v_1v_2v_3...v_n$, where $v_i \in V$, and $d_1(v_1) \uparrow$ or $d_1(v_1) \in A$, then*
$$d_1\nabla_a^v d_2 \cong d_1\nabla_a[v_1 \mapsto [v_2 \mapsto ... \mapsto [v_n \mapsto d_2]...].$$

(3) 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:

$$d_1\nabla_a^v d_2 \cong d_1\nabla_a(\Rightarrow v(d_2)).$$

7 Algebras of Nominative Data

The operations on nominative data of different classes defined earlier give a rise to algebraic structures defined below. We will focus on the types TND_{CS}, TND_{SC}, TND_{CC}, because the type TND_{SS} is rather simple and it is straightforward to define an algebra on it.

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

Definition 5. *An algebra of nominative data of the type TND_{SC} is an algebra*

$$NDA_{SC}(V,A) =< ND(V,A); (v \Rightarrow_a)_{v \in V^+}, (\Rightarrow v)_{v \in V^+}, (\nabla_a^v)_{v \in V^+} >$$

with the carrier set $ND(V,A)$ (data) and the following operations:

– *a family of partial unary associative denaming operations* $v \Rightarrow_a$: $ND(V,A) \tilde{\rightarrow} ND(V,A)$, $v \in V^+$;
– *a family of unary complex naming operations* $\Rightarrow v : ND(V,A) \rightarrow ND(V,A)$, $v \in V^+$;
– *a family of partial binary local overlapping operations* $\nabla_a^v : ND(V,A) \times ND(V,A) \tilde{\rightarrow} ND(V,A)$, $v \in V^+$.

Definition 6. *An algebra of nominative data of the type TND_{CS} is an algebra*

$$NDA_{CS}(V,A) =< NDVS(V,A); (v \Rightarrow_a)_{v \in V^+}, (\Rightarrow v)_{v \in V^+}, (\nabla_a^v)_{v \in V^+} >$$

with the carrier set $NDVS(V,A)$ (data) and the following operations:

– *a family of partial unary associative denaming operations* $v \Rightarrow_a$: $NDVS(V,A) \tilde{\rightarrow} NDVS(V,A)$, $v \in V^+$;
– *a family of unary complex naming operations* $\Rightarrow v : NDVS(V,A) \rightarrow NDVS(V,A)$, $v \in V^+$;
– *a family of partial binary local overlapping operations* $\nabla_a^v : NDVS(V,A) \times NDVS(V,A) \tilde{\rightarrow} NDVS(V,A)$, $v \in V^+$.

Definition 7. *An algebra of nominative data of the type TND_{CC} is an algebra*

$$NDA_{CC}(V,A) =< NDVC(V,A); (v \Rightarrow_a)_{v \in V^+}, (\Rightarrow v)_{v \in V^+}, (\nabla_a^v)_{v \in V^+} >$$

with the carrier set $NDVC(V,A)$ (data) and the following operations:

– *a family of partial unary associative denaming operations* $v \Rightarrow_a$: $NDVC(V,A) \tilde{\rightarrow} NDVC(V,A)$, $v \in V^+$;
– *a family of unary complex naming operations* $\Rightarrow v : NDVC(V,A) \rightarrow NDVC(V,A)$, $v \in V^+$;
– *a family of partial binary local overlapping operations* $\nabla_a^v : NDVC(V,A) \times NDVC(V,A) \tilde{\rightarrow} NDVC(V,A)$, $v \in V^+$.

8 Nominative Equivalence

Complex-named data have an inherent associated hierarchical naming structure. However, from the viewpoint of their information content that can be obtained from such data using the operation of associative denaming, some pairs of complex-named data like $[v_1 \mapsto [v_2 \mapsto [v_3 \mapsto 1]]]$ and $[v_1 v_2 v_3 \mapsto 1]$ that have different hierarchical naming structure are essentially equivalent.

The following equivalence relation on complex-named data called nominative equivalences formalizes this observation.

Definition 8 (Paths and terminal paths in data).

(1) A path in a complex-named data $d \in NDVC(V, A)$ is a nonempty sequence $(v_1, v_2, ..., v_n)$ of words from V^+ such that $((d(v_1))(v_2)...)(v_n)$ is defined. The value $((d(v_1))(v_2)...)(v_n)$ is called the value of the path $(v_1, v_2, ..., v_n)$ in d.
(2) A path in a complex-named data $d \in NDVC(V, A)$ is called a terminal path, if its value in d belongs to $A \cup \{\emptyset\}$.

Definition 9 (Nominative inclusion and equivalence).

(1) A complex-named data $d_1 \in NDVC(V, A)$ is nominatively included in a complex-named data $d_2 \in NDVC(V, A)$, 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 exists 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) Two complex-named data $d_1, d_2 \in NDVC(V, A)$ are nominative equivalent (denoted as $d_1 \approx d_2$), if d_1 is nominatively included in d_2 and d_2 is nominatively included in d_1.

It is known [5] that nominative inclusion is a preorder on $NDVC(V, A)$ and nominative equivalence is an equivalence relation on $NDVC(V, A)$.

Nominative equivalent data may have different hierarchical naming structures, but they turn into the same TND_{CS} data when they are flattened, e.g.:

- $[v_1 \mapsto [v_2v_3 \mapsto 1, v_2v_4 \mapsto 2]] \approx [v_1v_2v_3 \mapsto 1, v_1v_2v_4 \mapsto 2]$;
- $[v_1v_2 \mapsto [v_3 \mapsto 1, v_4 \mapsto 2]] \approx [v_1v_2v_3 \mapsto 1, v_1v_2v_4 \mapsto 2]$.

Theorem 1. *Nominative equivalence is a congruence on $NDA_{CC}(V, A)$.*

Proof. From [5] it follows that for each pair of $d_1, d_2 \in NDVC(V, A)$ the operations of naming and associative denaming satisfy the following property (called *nominative stability*): if $d_1 \approx d_2$ and an operation is defined on d_1, then it is defined on d_2 and its values on d_1 and d_2 are nominative equivalent. Similarly, from [5] it follows that if $v \in V^+$, $d_1, d_2 \notin A$, $d_1 \approx d'_1$, $d_2 \approx d'_2$, then $d_1 \nabla^v_a d_2 \approx d'_1 \nabla^v_a d'_2$. This implies that nominative equivalence is a congruence on $NDA_{CC}(V, A)$. □

This result allows us to construct a factor (quotient) algebra of $NDA_{CC}(V, A)$.

Let us denote by $NDA^{\approx}_{CC}(V, A)$ the factor algebra of $NDA_{CC}(V, A)$ by the nominative equivalence $\approx$.

Lemma 1. *$NDA_{SC}(V, A)$ is isomorphic to $NDA^{\approx}_{CC}(V, A)$.*

Proof (Sketch). Let us define a function $\iota^{\approx}_{13} : ND(V, A) \to 2^{NDVC(V,A)}$ such that $\iota^{\approx}_{13}(d) = \{d' \in NDVC(V, A) \mid d' \approx d\}$ (note that $ND(V, A) \subset NDVC(V, A)$). It is easy to show that each class of nominative equivalence has a unique member from $ND(V, A)$ and the operations on equivalence classes correspond to the operations on TND_{CS} data under this mapping. Then $\iota^{\approx}_{13}$ is an isomorphism from $NDA_{CS}(V, A)$ to $NDA^{\approx}_{CC}(V, A)$. □

Lemma 2. *$NDA_{CS}(V, A)$ is isomorphic to $NDA^{\overset{\sim}{\approx}}_{CC}(V, A)$.*

Proof (Sketch). Let us define a function $\iota^{\overset{\sim}{\approx}}_{23} : NDVS(V, A) \to 2^{NDVC(V,A)}$ as follows: $\iota^{\overset{\sim}{\approx}}_{23}(d) = \{d' \in NDVC(V, A) \mid d' \approx d\}$ (note that $NDVS(V, A) \subset NDVC(V, A)$). It is easy to show that each class of nominative equivalence has a unique member from $NDVS(V, A)$ and the operations on equivalence classes correspond to the operations on TND_{SC} data under this mapping. Then $\iota^{\overset{\sim}{\approx}}_{23}$ is an isomorphism from $NDA_{SC}(V, A)$ to $NDA^{\overset{\sim}{\approx}}_{CC}(V, A)$. $\square$

Theorem 2. *Algebras $NDA_{SC}(V, A)$, $NDA_{CS}(V, A)$, and $NDA^{\overset{\sim}{\approx}}_{CC}(V, A)$ are isomorphic.*

Proof. Follows immediately from Lemmas 1 and 2. $\square$

Let us denote by $D_3/\approx$ the carrier set of $NDA^{\overset{\sim}{\approx}}_{CC}(V, A)$.

Let $\iota^{\overset{\sim}{\approx}}_{13}$ and $\iota^{\overset{\sim}{\approx}}_{23}$ be the isomorphisms defined in the proofs of Lemmas 1 and 2. We will call $(\iota^{\overset{\sim}{\approx}}_{13})^{-1}$ the *layering* operation and $(\iota^{\overset{\sim}{\approx}}_{23})^{-1}$ the *flatting* operation.

For each $d \in D_3$ let $\iota^{\overset{\sim}{\approx}}_{3}(d)$ be the $\approx$-equivalence class of d.

Let us introduce the following inclusion maps for classes of nominative data: $\iota_{01} : D_0 \hookrightarrow D_1$, $\iota_{02} : D_0 \hookrightarrow D_2$, $\iota_{13} : D_1 \hookrightarrow D_3$, $\iota_{23} : D_2 \hookrightarrow D_3$.

$$\begin{tikzcd}
 & D_0 \arrow[dl, hook, "\iota_{01}"'] \arrow[dr, hook, "\iota_{02}"] & \\
D_1 \arrow[r, "\iota^{\overset{\sim}{\approx}}_{13}"] \arrow[dr, hook, "\iota_{13}"'] & D_3/\approx & D_2 \arrow[l, "\iota^{\overset{\sim}{\approx}}_{23}"'] \arrow[dl, hook, "\iota_{23}"] \\
 & D_3 \arrow[u, "\iota^{\overset{\sim}{\approx}}_{3}"'] &
\end{tikzcd}$$

Fig. 1. Classes of nominative data and mappings between them.

Then it is easy to check that the diagram shown in Fig. 1 is commutative. This diagram illustrates the types of nominative data and mappings between them.

9 Compositions of Functions and Predicates over Nominative Data

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

Denote

$$Pr_{CC}(V, A) = NDVC(V, A) \mathrel{\tilde{\to}} \{T, F\},$$

$$Fn_{CC}(V, A) = NDVC(V, A) \mathrel{\tilde{\to}} NDVC(V, A),$$

where T and F denote logical constants (true and false). We will assume that they 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.*

Similarly, let us denote the following sets of functions and predicates on data of the types TND_{SC} and TND_{CS}:

$$Pr_{SC}(V, A) = ND(V, A) \tilde{\rightarrow} \{T, F\},$$

$$Fn_{SC}(V, A) = ND(V, A) \tilde{\rightarrow} ND(V, A),$$

$$Pr_{CS}(V, A) = NDVS(V, A) \tilde{\rightarrow} \{T, F\},$$

$$Fn_{CS}(V, A) = NDVS(V, A) \tilde{\rightarrow} NDVS(V, A).$$

Let us consider conventional logical and programming compositions of predicates and functions. These compositions are defined analogously for functions and predicates over each type of nominative data, so below we give a common definition in which the symbol Fn should be understood as one of Fn_{CC}, Fn_{CS}, or Fn_{SC}, and the symbol Pr should be understood as one of Pr_{CC}, Pr_{CS}, Pr_{SC} However, within each definition of a composition below all occurrences of the symbols Fn, Pr refer to sets of functions and predicates over nominative data of the same type, i.e. either TND_{CC}, or TND_{CS}, or TND_{SC}.

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)

$$\bullet : Fn(V, A) \times Fn(V, A) \rightarrow Fn(V, A)$$

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 [12] (denoted using the infix notation)

$$\cdot : Fn(V, A) \times Pr(V, A) \rightarrow Pr(V, A)$$

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 $Asg^u : Fn(V, A) \rightarrow Fn(V, A)$ with a parameter $u \in V^+$ is defined as follows: for each $f \in Fn(V, A)$ and data d,

$$(As^u(f))(d) \cong d \nabla_a^u f(d)$$

(where ∇_a^u refers to the local overlapping on the corresponding type of nominative data).

- The composition of superposition into a function

$$S_F^{u_1,u_2,\ldots,u_n} : Fn(V,A) \times (Fn(V,A))^n \to Fn(V,A)$$

  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:

$$S_F^{u_1,u_2,\ldots,u_n}(f, f_1, ..., f_n)(d) \cong f(...((d\nabla_a^{u_1} f_1(d))\nabla_a^{u_2} f_2(d))...\nabla_a^{u_n} f_n(d))...).$$

  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 $S_F^{u_1,u_2,\ldots,u_n}$.
- The composition of superposition into a predicate

$$S_P^{u_1,u_2,\ldots,u_n} : Pr(V,A) \times (Fn(V,A))^n \to Pr(V,A)$$

  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:

$$S_P^{u_1,u_2,\ldots,u_n}(p, f_1, ..., f_n)(d) \cong p(...((d\nabla_a^{u_1} f_1(d))\nabla_a^{u_2} f_2(d))...\nabla_a^{u_n} f_n(d))...).$$

  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 $S_P^{u_1,u_2,\ldots,u_n}$.
- Branching composition $IF : Pr(V,A) \times Fn(V,A) \times Fn(V,A) \to Fn(V,A)$ is defined as follows: for each $p \in Pr(V,A)$ and $f, g \in Fn(V,A)$:

$$IF(p,f,g)(d) \cong \begin{cases} f(d), & \text{if } p(d) \downarrow = T; \\ g(d), & \text{if } p(d) \downarrow = F; \\ \text{undefined}, & \text{if } p(d) \uparrow . \end{cases}$$

- 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 data 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 $f^{(n)}$ denotes a n-times sequential composition of f with itself (assuming that $f^{(0)}$ 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:

$$(\neg p)(d) \cong \begin{cases} T, & \text{if } p(d) \downarrow = F; \\ F, & \text{if } p(d) \downarrow = T; \\ \text{undefined}, & \text{if } p(d) \uparrow . \end{cases}$$

- 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) $u! \in Pr(V, A)$ with a parameter $u \in V^+$ is defined as follows:

$$u!(d) = \begin{cases} T, & \text{if } u \Rightarrow_a (d) \downarrow; \\ F, & \text{if } u \Rightarrow_a (d) \uparrow. \end{cases}$$

- Empty constant function (null-ary composition) $Empty \in Fn(V, A)$ is defined as follows: $Empty(d) = \emptyset$ for all data d.
- Emptiness checking predicate (null-ary composition) $IsEmpty \in Fn(V, A)$ is defined as follows:

$$IsEmpty(d) = \begin{cases} T, & \text{if } d = \emptyset; \\ F, & \text{if } d \neq \emptyset. \end{cases}$$

The compositions defined above allow us to specify a rather expressive program language. This language generalizes several simple program algebras used in the approaches such as Glushkov Algorithmic Algebras [12], Floyd-Hoare logics [8,9], algorithmic logics [14], dynamic logics [15], etc. In Glushkov Algorithmic Algebras predicates and functions are considered as partial as we do here. Therefore we consider our approach as a generalization of Glushkov Algorithmic Algebras.

Definition 10. *An Associative Nominative Glushkov Algorithmic Algebra of predicates and functions over the nominative data of the type TND_{CC} is a two-sorted algebra*

$$NGA^a_{CC}(V, A) =< Pr_{CC}(V, A), Fn_{CC}(V, A); \vee, \neg, \bullet, IF, WH, \cdot, (Asg^u)_{u \in V^+},$$

$$(S^{\bar{u}}_F)_{\bar{u} \in \bar{U}}, (S^{\bar{u}}_P)_{\bar{u} \in \bar{U}}, Id, True, \bot_F, \bot_P, (u!)_{u \in V^+}, Empty, IsEmpty >$$

with the carrier sets $Pr_{CC}(V, A)$ (predicates) and $Fn_{CC}(V, A)$ (functions) for some V and A with the following operations: the disjunction $\vee$ and the negation $\neg$ compositions on predicates, the sequential composition of functions $\bullet$, the branching composition IF, the cycle composition WH, the prediction composition $\cdot$, the family of assignment compositions Asg^u, $u \in V^+$, the families of superposition compositions $S^{\bar{u}}_P$, $S^{\bar{u}}_P$ for $\bar{u} \in \bar{U}$, the identity composition on functions Id, the true constant predicate $True$, and the following constant elements of the carrier sets (null-ary compositions): the bottom predicate $\bot_P$, the bottom function $\bot_F$, the family of name checking predicates $u!$, $u \in V^+$, the empty constant function $Empty$, and the emptiness checking predicate $IsEmpty$.

Similarly, we can define algebras of functions and predicates over nominative data of the types TND_{CS} and TND_{SC}:

Definition 11.

(1) An Associative Nominative Glushkov Algorithmic Algebra of predicates and functions over the nominative data of the type TND_{SC} is a two-sorted algebra

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

(2) An Associative Nominative Glushkov Algorithmic Algebra of predicates and functions over the nominative data of the type TND_{CS} is a two-sorted algebra

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

10 Nominative Stability

The type TND_{CC} of nominative data is the most rich and interesting among other types of nominative data, so we will focus on it.

The binary relation of *nominative stability* is a formalization of the idea that a program's behavior does not change, if the hierarchical naming structure of its data changes. 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. However, it should be noted that many practical programming languages like C++ and Java do not have this feature. This gives a rise to the following informal question: *which properties a programming language must satisfy in order to guarantee that its programs behave correctly regardless of the hierarchical naming structure of their data*?

We formalize such a notion of independence of a program behavior from the hierarchical naming structure of data as nominative stability of functions and predicates.

Definition 12. *A predicate $p \in Pr_{CC}(V,A)$ is called nominative stable, if for each $d_1, d_2 \in NDVC(V,A)$, if $p(d_1)\downarrow$ and $d_1 \approx d_2$, then $p(d_2)\downarrow$ and $p(d_1) = p(d_2)$.*

Definition 13. *A function $f \in Fn_{CC}(V,A)$ is called nominative stable, if for each $d_1, d_2 \in NDVC(V,A)$, if $f(d_1)\downarrow$ and $d_1 \approx d_2$, then $f(d_2)\downarrow$ and $f(d_1) \approx f(d_2)$.*

Let us denote by $PrNS(V, A)$ the set of all nominative stable predicates $p \in Pr_{CC}(V, A)$ and by $FnNS(V, A)$ the set of all nominative stable functions $f \in Fn_{CC}(V, A)$.

Theorem 3. *$PrNS(V, A)$ and $FnNS(V, A)$ form a sub-algebra of the Associative Nominative Glushkov Algorithmic Algebra $NGA^a_{CC}(V, A)$.*

Proof (Sketch). Using the same methods which were used in [5] to show monotonicity of program compositions it is straightforward to show that $FnNS(V, A)$ is closed under the sequential composition $\bullet$ and the identity composition Id, and if $p \in PrNS(V, A)$ and $f, g \in FnNS(V, A)$, then $f \cdot p \in PrNS(V, A)$, $IF(p, f, g)$ and $WH(p, f) \in FnNS(V, A)$. Also, it is trivial to show that $PrNS(V, A)$ is closed under the negation and disjunction compositions.

From [5, Lemmas 7.3 and 7.4] it follows that the assignment composition preserves nominative stability, i.e. if a function $f \in FnN(V, A)$ is nominative stable, then $Asg^u(f)$ is nominative stable, so if $f \in FnNS(V, A)$, then $Asg^u(f) \in FnNS(V, A)$.

Let us show that superposition into a function preserves nominative stability. Let $n \geq 1$, $f, f_1, ..., f_n \in FnN(V, A)$, and $(u_1, u_2, ..., u_n) \in \bar{U}$. Let us show that $S_F^{u_1,...,u_n}(f, f_1, ..., f_n) \in FnNS(V, A)$. Let $d_1, d_2 \in NDVC(V, W)$, $S_F^{u_1,...,u_n}(f, f_1, ..., f_n)(d_1) \downarrow$, and $d_1 \approx d_2$. Then $d_1 \notin A$ for all $i \in \{1, 2, ..., n\}$. Thus $d_2 \notin A$ and for all $i \in \{1, 2, ..., n\}$, $f_i(d_2) \approx f_i(d_1)$. Then

$$(...((d_2 \nabla_a^{u_1} f_1(d_2)) \nabla_a^{u_2} f_2(d_2))... \nabla_a^{u_n} f_n(d_2))...) \downarrow$$

and

$$(...((d_1 \nabla_a^{u_1} f_1(d_1)) \nabla_a^{u_2} f_2(d_1))... \nabla_a^{u_n} f_n(d_1))...) \approx$$
$$\approx (...((d_2 \nabla_a^{u_1} f_1(d_2)) \nabla_a^{u_2} f_2(d_2))... \nabla_a^{u_n} f_n(d_2))...),$$

because $\approx$ is a congruence in $NDA_{CC}(V, A)$. Taking into account that $f \in FnNS(V, A)$, we conclude that

$$S_F^{u_1,u_2,...,u_n}(f, f_1, ..., f_n)(d_2) \downarrow$$

and

$$S_F^{u_1,u_2,...,u_n}(f, f_1, ..., f_n)(d_2) \approx S_F^{u_1,u_2,...,u_n}(f, f_1, ..., f_n)(d_1).$$

Thus $S_F^{u_1,u_2,...,u_n}(f, f_1, ..., f_n) \in FnNS(V, A)$.

Similarly it is straightforward to show that superposition into a predicate preserves nominative stability.

Besides, it is easy to check that

$$\{True, \bot_P, IsEmpty\} \cup \{u! \mid u \in V^+\} \subset PrNS(V, A)$$

and

$$\{\bot_F, Empty\} \subset FnNS(V, A).$$

We conclude that $PrNS(V, A)$ and $FnNS(V, A)$ are closed under all compositions of $NGA^a_{CC}(V, A)$, so they form a sub-algebra. □

Let us define the following equivalence relation on $FnNS(V, A)$:

$$f \equiv_F g$$

(where $f, g \in FnNS(V, A)$), if $d_1 \approx d_2$ and $f(d_1) \downarrow$ implies $f(d_1) \approx g(d_2)$.

Also, let us define the following equivalence relation on $PrNS(V, A)$:

$$p_1 \equiv_P p_2$$

(where $p_1, p_2 \in PrNS(V, A)$), if $d_1 \approx d_2$ and $p(d_1) \downarrow$ implies $p_1(d_1) = p_2(d_2)$.

Let

$$NGANS^a(V, A) =$$

$$< PrNS(V, A), FnNS(V, A); \vee, \neg, \bullet, IF, WH, \cdot, (Asg^u)_{u \in V^+},$$

$$(S_F^{\bar{u}})_{\bar{u} \in \bar{U}}, (S_P^{\bar{u}})_{\bar{u} \in \bar{U}}, Id, True, \bot_F, \bot_P, (u!)_{u \in V^+}, Empty, IsEmpty >$$

be the sub-algebra mentioned in Theorem 3.

Theorem 4. *The relations $\equiv_F$, $\equiv_P$ are congruences on $NGANS^a(V, A)$.*

We omit the proof, as it can be easily obtained from Theorem 1.

This theorem allows us to construct a factor algebra.

Let $NGANS^a_{\equiv}(V, A)$ be the factor algebra of $NGANS^a(V, A)$ by the congruences $\equiv_F$ and $\equiv_P$.

Theorem 5. *The algebras $NGA^a_{CS}(V, A)$, $NGA^a_{SC}(V, A)$, and $NGANS^a_{\equiv}(V, A)$ are isomorphic.*

Proof (Sketch). Using Lemmas 1 and 2 it is straightforward to show that the algebra $NGANS^a_{\equiv}(V, A)$ is isomorphic to $NGA^a_{CS}(V, A)$ and $NGA^a_{SC}(V, A)$. Thus $NGA^a_{CS}(V, A)$ and $NGA^a_{SC}(V, A)$ are isomorphic. □

The obtained result has several interpretations and possible applications. Firstly, a programmer can construct a nominative stable program oriented on a certain hierarchical naming structure of input data, but this program would give equivalent results, if input data were changed to equivalent data. Such stability simplifies programming with complex data making it "softer" because the programmer should not remember the current structure of data. Secondly, the class of such programs can be correctly implemented using nominative data from the class TND_{SC} (or TND_{CS}) only. Third, the obtained result gives a perspective for reduction of formulas of a logic over hierarchical nominative data to formulas of a logic over "flat" data, which is closer to classical logic.

11 Conclusions and Future Work

We have investigated basic properties of nominative functions and predicates from the perspective of the abstract algebra. We have defined basic compositions of such functions and predicates and introduced a two-sorted algebra which generalizes Glushkov algorithmic algebras. An advantage of this generalization is that its functions are defined over nominative data which give a natural and adequate representation of data structures commonly used in programming (as we have demonstrated in Sect. 6). We have considered an equivalence relation on nominative data with complex names and the notions of nominative stability of nominative functions and predicates and proved the set of a nominative stable functions and the set of nominative stable predicates form its sub-algebra.

In the forthcoming papers we plan to continue our study of theoretical aspects of composition-nominative approach and describe practical applications of this approach.

References

1. Sannella, D., Tarlecki, A.: Foundations of Algebraic Specification and Formal Software Development. Springer, Heidelberg (2012)
2. Nikitchenko, N.S.: A composition-nominative approach to program semantics. Technical report, IT-TR 1998–020, Technical University of Denmark (1998)
3. Nikitchenko, M., Tymofieiev, V.: Satisfiability in composition-nominative logics. Cent. Eur. J. Comput. Sci. **2**, 194–213 (2012)
4. Kryvolap, A., Nikitchenko, M., Schreiner, W.: Extending Floyd-Hoare logic for partial pre- and postconditions. In: Ermolayev, V., Mayr, H.C., Nikitchenko, M., Spivakovsky, A., Zholtkevych, G. (eds.) ICTERI 2013. CCIS, vol. 412, pp. 355–378. Springer, Heidelberg (2013)
5. Nikitchenko, M., Ivanov, I.: Programming with nominative data. In: Proceedings of CSE'2010 International Scientific Conference on Computer Science and Engineering, Kosice, Slovakia, 20–22 September 2010, pp. 30–39 (2010)
6. Nikitchenko, M., Ivanov, I.: Composition-nominative languages of programs with associative denaming. Visnyk (Bulletin) of the Lviv University Ser. Appl. Math. Inform. **16**, 124–139 (2010)
7. Nielson, H.R., Nielson, F.: Semantics with Applications: A Formal Introduction. John Wiley & Sons Inc., New York (1992)
8. Floyd, R.: Assigning meanings to programs. Proc. Am. Math. Soc. Symp. Appl. Math. **19**, 19–31 (1967)
9. Hoare, C.: An axiomatic basis for computer programming. Commun. ACM **576–580**, 583 (1969)
10. Skobelev, V.: Local algorithms on graphs. Publishing house of Institute of Applied Mathematics and Mechanics of NAS of Ukraine (in Russian) (2003)
11. Roman, S.: Lattices and Ordered Sets. Springer, New York (2008)
12. Glushkov, V.: Automata theory and formal transformations of microprograms. Cybernetics (in Russian) **5**, 3–10 (1965)
13. Coquand, T., Paulin, C.: Inductively defined types. In: Martin-Löf, P., Mints, G. (eds.) COLOG 1988. LNCS, vol. 417, pp. 50–66. Springer, Heidelberg (1990)
14. Mirkowska, G., Salwicki, A.: Algorithmic Logic. Springer, New York (1987)
15. Harel, D., Kozen, D., Tiuryn, J.: Dynamic Logic. The MIT Press, Cambridge (2000)