Abstract—In the paper we describe a formalization of the notion of a nominative data with simple names and complex values in the Mizar proof assistant. Such data can be considered as a partial variable assignment which allows arbitrarily deep nesting and can be useful for formalizing semantics of programs that operate in real time environment and/or process complex data structures and for reasoning about the behavior of such programs.

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 talk “The Verifying Compiler: a Grand Challenge for computing research of the 21st century” [1] with the implication that an important step towards solution of this challenge is development of a verifying compiler that should have a high impact on software quality and reliability. More generally, one may argue that development of practical tools and methods of automatic static analysis of a program (e.g. model checking, verification against a formal specification using logical methods and theorem provers, etc.) that can make sure that it has the desired runtime properties before the program is run is an important research topic.

However, nowadays software is used in many application domains and the traditional idea of proving properties of the input-output relation associated with a program is not always sufficient.

For example, practically relevant safety properties of software in a real-time embedded system [2], [3] (that is a part of a larger hardware-software system which interacts with the physical environment using sensors and actuators) or a cyber-physical system [2], [3], [4] (e.g. that consist of networks of computing devices that interact with physical environment) normally can be expressed not in the form of a property of a program or its input-output relation itself, but in terms of admissible behaviors of a larger software-hardware system and an environment to which it belongs.

The way of proving such properties depends on the chosen mathematical model of the hardware and the environment and it is important to note that such a proof is practically relevant only under the assumption that the model relative to which safety is proven adequately represents the behavior of the real system and its environment, and the environment usually includes unknown and/or random elements, and the scope of the model usually is limited.

As a toy example, suppose that a program $P$ can control the behavior of a physical system $S$ by assigning and modifying the value of a certain parameter $p$. The behavior of $S$ is described by a differential equation $\frac{d}{dt}x(t) = f(p, t, x(t))$, where $f$ is some fixed real-valued function. Assume that:

- $p$ is updated by $P$ at discrete time moments $t = t_1, t_2, \ldots$ determined by $P$, between which $p$ remains constant (so on each bounded closed time interval $p$ can be considered a piecewise constant function);
- $P$ controls $S$ in the open-loop fashion, i.e. $P$ has no or does not use feedback from $S$.

Then one may formalize the above mentioned equation as a switched system [5] where $p$ is a switching signal (if $P$ uses some feedback from $S$ it may be considered as a kind of a hybrid dynamical system [6]) and assume that its solutions $t \mapsto x(t)$ defined on intervals of the form $[0, T)$, $T \in (0, +\infty) \cup \{+\infty\}$ describe all possible evolutions of the state of the system in continuous time which start at the initial time moment $t = 0$.

Suppose that we want to check that the composite (“cyber-physical”) system consisting of $S$ together with the program $P$ and a computing device which executes it, which we will denote as “$S + P$”, has the following property which we will call “$\text{NONNEG}$”: if $x(0) \geq 0$, then $x(t) \geq 0$ for all real $t \geq 0$ for any solution $x$ which is defined at $t = 0$ and cannot be continuously extended forward in time (i.e. the value $x$ that describes some characteristic of the system $S$ does not fall below 0, if it starts at or above 0).

In order to check, if it holds for a particular $f$ and $P$, it is necessary to formulate precisely the semantics of $P$.

For example, suppose that $f$ has a simple polynomial form $f(u, v, w) = uv^2$, so that the equation has the form $\frac{d}{dt}x(t) = p(t)x^2(t)$, and the program $P$ is given in the source code form in some imperative programming language with C-like syntax as in Algorithm I (where $L1\ldots L6$ are labels).

A common way of giving an operational semantics [7] to programs in languages of this type is to define the notion of a program state that includes the information about its current point of execution and the current content of its variables, and
define a state transition system that describes the possible paths of evolution ("runs") of the program state during execution. Then reasoning about the relation between the program states at different program execution points can be done using e.g. the Floyd-Hoare logic [8], [9], [10].

The content of variables in a program state is usually formalized as a function mapping names of program variables to their values (variable assignment). The commonly used notation for such an assignment has the form \([\text{var}_1 \mapsto \text{value}_1, \text{var}_2 \mapsto \text{value}_2, \ldots]\), where \(\text{var}_i\) are variables and \(\text{value}_i\) are the values that the variables have. In \(P\) we can consider \(i\) a normal read/write variable the value of which is stored in the memory or a CPU register. The variable \(p\) and the assignment to it can have different interpretations (e.g. normal memory location, a write-only register/hardware port, etc.).

Using the labels L1–L6 to identify program execution points, and variable assignments to represent the content of variables, a program state can be formalized as a pair "(label, variable assignment)”, and a possible run of \(P\) can have the form of a sequence such as:

\[
(L1, [i \mapsto 0, p \mapsto 0]), (L2, [i \mapsto 0, p \mapsto 0]), (L3, [i \mapsto 1, p \mapsto 0]), \ldots
\]

Obviously, in our case \(\text{NONNEG}\) cannot be checked by looking at such runs of \(P\) alone, instead the behavior of \(S\) and the timing of interaction between \(P\) and \(S\) should be taken into account. In particular, runs of \(P\) should be augmented with timing information to obtain a switching signal \(p\) that determines the behavior of \(S\) and ultimately allows one to check if \(\text{NONNEG}\) holds for \(S + P\).

A convenient way to add timing information and the behavior of \(S\) to runs of \(P\) is to extend the program state, or more specifically, extend the variable assignment with variables that represent time \((t)\) and the state of \(S\) \((x)\), although, or course, they differ in nature from \(i\) and \(p\). Ignoring the execution time of instructions other than "sleep(1)”, this extended idealized run of \(S + P\) can have the form:

\[
(L1, [i \mapsto 0, p \mapsto 0, t \mapsto 0, x \mapsto x_0]), (L2, [i \mapsto 0, p \mapsto 0, t \mapsto 0, x \mapsto x_0]), (L3, [i \mapsto 0, p \mapsto 0, t \mapsto 0, x \mapsto x_0]), \ldots
\]

where \(x_i = x(i)\) for a solution \(x\) of the switched system

\[
\frac{dx(t)}{dt} = p(t)x^2(t), x(0) = x_0,
\]

\[
p(t) = \begin{cases} 
  i, & t \in [i, i + 1), i \in \{0, 1, \ldots, 9\}, \\
  9, & t \geq 10.
\end{cases}
\]

Note that if \(x_0 \neq 0\), this solution \(x\) dominates the solution \(y\) of the initial value problem \(\frac{dy(t)}{dt} = y^2(t), y(1) = x_0\) for \(t \geq 1\), which is \(y(t) = \frac{1}{1 + x_0^{1/t}} - t\) and which has a finite time blow-up at \(t = 1 + x_0^{-1}\) (i.e. \(\lim_{t \to x_0^{-1}} x(t) = \infty\)), so \(x\) is undefined (and cannot be continuously extended) past the time \(1 + x_0^{-1}\). The physical meaning of this situation is model-and application-specific, e.g. it may represent a physical event after which the current model cannot represent adequately \(S\), or may be a modeling artifact. Some results and discussion of the physical meaning of finite time blow-up situations can be found e.g. in [11], [12], [13], [14].

In any case, such situations cannot be ignored during model analysis and property checking and has to be represented in a run of \(S + P\). E.g., although if \(x_0 \geq 0\), then \(x(t) \geq 0\) on the maximal interval of its existence, which is bounded from above, \(\text{NONNEG}\) does not hold since it requires \(x(t) \geq 0\) to hold for all \(t \geq 0\) for such an \(x\).

Although the model of \(S\) may lose its meaning after \(t = 1 + x_0^{-1}\), depending on the model and application, the run of a program \(P\) may still have sense past this time moment (e.g. if \(S\) represents a physical system remotely controlled by a computer running \(P\), a finite time blow-up indicates a critical failure in \(S\), then \(P\) may continue running after this event).

Naturally, the situation can be represented by partial variable assignments in the extended run of \(S + P\), e.g.:

\[
(L1, [i \mapsto 0, p \mapsto 1, t \mapsto 2]),
\]

means that \(x\) is genuinely undefined, but other variables are defined, meaningful and have assigned values at time \(t = 2\).

Basically, a partial variable assignment \(d\) is a set of pairs of names and values, where names are chosen from some fixed set \(V\), but not all elements of \(V\) may appear in \(d\) as names (e.g. \(V = \{i, p, t, x\}, d = [i \mapsto 0, p \mapsto 1, t \mapsto 2]\)). Such assignments are also called nominative sets [15].

In contrast, a variable assignment can be called total, if all elements of \(V\) appear in it as names (e.g. \(V = \{i, p, t, x\}, d = [i \mapsto 0, p \mapsto 1, t \mapsto 2, x \mapsto 3]\)).

Obviously, total variable assignments can be formalized mathematically as total functions on \(V\) and partial assignments (nominative sets) can be formalized as partial functions on \(V\).

Reasoning about total variable assignments is well supported in tools that aid formal specification and verification of software, e.g. proof assistants [16] such as Isabelle, Coq, PVS, etc. In particular, Isabelle/HOL “record” data types are convenient for introducing total assignments with a predefined set of names. Otherwise, they may be formalized as functions on a type of names.

On the other hand, partial variable assignments are usually implemented manually on top of total assignments using option types (data types that add a special “none”/undefined value to an existing type), and a built-in library of basic
operations on them that reflect program operations is generally not available. Another issue is that built-in or library data types that represent partial variable assignments with (arbitrarily deep) nesting such as

\[
\begin{align*}
\text{local} & \text{vars} \rightarrow [i \mapsto 1, j \mapsto 0], \\
\text{IO} & \text{interface} \rightarrow [p \mapsto 1], \\
\text{global} & \text{time} \rightarrow 1, \\
\text{systemS} & \text{state} \rightarrow [x \mapsto 1], \\
\text{another} & \text{SystemState} \rightarrow \\
& \begin{cases} 
1 & \text{subsystem1} \rightarrow [y \mapsto 1, z \mapsto 1], \\
2 & \text{subsystem2} \rightarrow [y \mapsto 1, z \mapsto 1] \end{cases}
\end{align*}
\]

are also not available (i.e. partial variable assignments where the values of some variables themselves can be partial variable assignments). However, such assignments can be very useful to reflect the typical way in which data is grouped in programs and formalize programs that operate on complex data structures, e.g. such partial assignments naturally formalize data encoded in the popular JSON (JavaScript Object Notation) data format widely used in web applications.

In this paper we propose a library (written in the Mizar proof assistant [17], [18]) that provides a formalization of nominative sets and operations on them. The Mizar system has its own proof verifier used to verify the logical correctness of proofs written in the Mizar language\(^1\). The system contains a very rich library (based on an axiomatic set theory [29]) of formalized mathematical theories called Mizar Mathematical Library (MML)\(^2\).\(^3\)\(^4\). It has also a library support for the notion of a partial function without using option types. It is then well-suited for formalizing the mentioned partial variable assignments with nesting, and, more generally, has a potential for developing formalizations of models of real-time and cyber-physical systems and logics for reasoning about them. We hope that it will be useful for formalizing models of hardware-software systems and applying logical methods to verification such systems. In this direction we developed an extension of the Floyd-Hoare logic [10] that takes advantage of partial variable assignments and also supports partially defined pre- and post-condition predicates. More information about it can be found in [10].

In our library we use the theory of nominative data [15], [37], [38] from the composition-nominative approach [37] to program semantics: partial variable assignments are formalized as so-called nominative data with simple (unstructured) names and complex (structured) values. They are also called Type SC (simple names, complex values) nominative data, or nominative data of the type \(TND_{SC} [15]\). They have hierarchical structure and can be considered as labeled oriented trees with arcs labeled with names.

The set of nominative data over a given set of (simple/basic) names \(V\) and set of atomic (basic) values \(A\) is denoted as \(ND(V,A)\) and is defined as follows [15]:

\[
ND(V,A) = \bigcup_{k \geq 0} ND_k(V,A),
\]

where

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

\[
ND_{k+1}(V,A) = A \cup \{V \rightarrow ND_k(V,A)\}, \quad k \geq 0.
\]

where for any set \(X\), \(V \rightarrow X\) denotes the set of all partial functions from \(V\) to \(X\), the domain of which is a finite set, and \(\emptyset\) is the empty set which is considered to be the empty nominative data and alternatively denoted as \([\]\). The elements of \(V \rightarrow X\) are also called nominative data with simple names and simple values, or Type SS nominative data over a set of names \(V\) and a set of values \(X\).

If \(v_1, v_2, \ldots, v_n \in V\) are pairwise different (simple) names, and \(a_1, a_2, \ldots, a_n\) are elements of \(ND(V,A)\), then \([v_1 \mapsto d_1, v_2 \mapsto d_2, \ldots, v_n \mapsto d_n]\) denotes the unique nominative data \(d \in ND(V,A)\) such that the domain of \(d\) (the set on which \(d\), as a function, is defined) is \(\{v_1, v_2, \ldots, v_n\}\) and \(d(v_j) = a_i\) for \(i = 1, 2, \ldots, n\).

For example, if \(a, b, c\) are distinct names (arbitrary objects), then \([a \mapsto [a \mapsto 1, b \mapsto 2]], c \mapsto 2] \in ND([a,b,c], \{1,2\})\).

The elements of \(A\) are called atomic nominative data, while the elements of \(ND(V,A)\)\(\backslash A\) are called non-atomic nominative data.

The basic operations on Type SC nominative data are:

- **Naming** – creating a nominative data of the form \([v \mapsto d]\) from a given nominative data \(d\) and a name \(v\) (this is denoted as \(v(d)\), or a nominative data of the form \([v_1 \mapsto v_2 \mapsto \ldots \mapsto v_n \mapsto d]\) from a given nominative data \(d\) and a finite sequence of names \(v_1, v_2, \ldots, v_n\) (this is denoted as \(v_1 v_2 \ldots v_n(d)\)). For example, if \(v_3, v_4\) are different names, then \(v_1 v_2 ([v_3 \mapsto 1, v_4 \mapsto 2]) = [v_1 \mapsto v_2 \mapsto [v_3 \mapsto 1, v_4 \mapsto 2]]\).

- **Denaming** – obtaining the value corresponding to a given name \(v\) or a sequence of names \(v_1, v_2, \ldots, v_n\) in a given non-atomic nominative data \(d\) (i.e. \(d \notin A\); if \(d \in A\), denaming is undefined on \(d\)). I.e., if \(d = [a_1 \mapsto a_1, a_2 \mapsto a_2, \ldots, a_m \mapsto a_m]\) and \(v = v_i\) for some \(i\), then \(a_i\) is the result of denaming the name \(v\) in \(d\) (it is denoted as \(v(d)\), so \(a_i = a_i(d)\); it is assumed that \(v(d)\) is undefined, if \(v\) is not among \(a_1, a_2, \ldots, a_m\)). For a sequence of names \(v_1, \ldots, v_n\), denaming of a nominative data \(d\) is denoted as \(v_1 v_2 \ldots v_n \Rightarrow_a(d)\) and has the following

---

\(^1\)Research on using specialized external systems to increase computational power of the Mizar system is also conducted [19], [20], [21].

\(^2\)The Mizar language is a declarative language designed to write mathematical documents. It contains rules for writing traditional mathematical items (e.g. definitions, theorems, proof steps, etc.). It also provides syntactic constructions to launch specialized algorithms (e.g. term identifications, term reductions [22], flexary connectives [23], delinitional expansions [24]) which increase the computational power of the verifier (e.g. equational calculus [25], [26], properties of functors and predicates [27], [28]).

\(^3\)MML contains developments on various domains of mathematics, including set theory, calculus, topology, lattice theory [30], group theory, category theory, algebra [31], rough sets [32], and others.

\(^4\)Because of the size, the MML is also a subject of research on optimization of its structure, including the improvement of legibility of proofs [33], [34], [35] and removing duplications of theorems and definitions [36].
We defined the notion of a Type SS nominative data (TypeSSNominativeData) as a nominative set with the finite graph:

\[
\text{registration}
\]

let \( V, A \) be set;

\[
\text{cluster finite for NominativeSet of } V, A;
\]

end;

\[
\text{definition}
\]

let \( V, A \) be set;

\[
\text{mode TypeSSNominativeData of } V, A
\]

\[
\text{is finite NominativeSet of } V, A;
\]

end;

We defined the set \( NDSS(V, A) \) of all Type SS nominative data as follows:

\[
\text{definition}
\]

let \( V, A \) be set;

\[
\text{func NDSS}(V, A) \rightarrow \text{set equals}
\]

the set of all \( D \) where \( D \) is TypeSSNominativeData of \( V, A \);

end;

and proved that it is nonempty for all sets \( V \) and \( A \):

\[
\text{registration}
\]

let \( V, A \) be set;

\[
\text{cluster NDSS}(V, A) \rightarrow \text{non empty};
\]

end;

The following definitions introduce the notion of a type SC nominative (TypeSCNominativeData), including nonatomic nominative data (NonatomicND).

\[
\text{definition}
\]

let \( V, A \) be set;

\[
\text{let } S \text{ be FinSequence;}
\]

\[
\text{pred } S \text{ ISNDRankSeq } V, A \text{ means}
\]

\[
S.1 = NDSS(V, A) \land
\]

for \( n \) being Nat st \( n \) in dom \( S \) & \( n + 1 \) in dom \( S \)

holds \( S.(n+1) = NDSS(V, A \setminus S\cdot n) \);

end;

\[
\text{definition}
\]

let \( V, A \) be set;

\[
\text{mode NonatomicND of } V, A \rightarrow \text{Function means}
\]

\[
\text{ex } S \text{ being FinSequence st } S \text{ ISNDRankSeq } V, A \land
\]

in Union \( S \);

end;

\[
\text{definition}
\]

let \( V, A \) be set;

\[
\text{mode TypeSCNominativeData of } V, A \rightarrow \text{set means}
\]

\[
it \text{ in } A \text{ or it is NonatomicND of } V, A;
\]

end;

\[
\text{definition}
\]

let \( V, A \) be set;

\[
\text{let } D \text{ be TypeSCNominativeData of } V, A;
\]

\[
\text{attr } D \text{ is atomicND means}
\]

\[
D \text{ in } A;
\]

\[
\text{attr } D \text{ is non-atomicND means}
\]

\[
D \text{ is NonatomicND of } V, A;
\]

end;

\[
\text{registration}
\]

let \( V \) be set; let \( A \) be non empty set;

\[
\text{cluster atomicND}
\]
for TypeSCNominativeData of V,A;
end;

registration
let V,A be set;
cluster non-atomicND
for TypeSCNominativeData of V,A;
end;

where FinSequence denotes a finite sequence [42], . (dot) denotes a function application [43], dom is the domain of the function [44], and Union denotes the union of the codomain of the function [45].

Then, we proved several theorems suitable for defining new IsNDRankSeq sequences:

- **Theorem 1**: For S being FinSequence st S IsNDRankSeq V,A
  for n being Nat st n in dom S holds (S|n) IsNDRankSeq V,A;

- **Theorem 2**: For S being FinSequence st S IsNDRankSeq V,A
  holds S ^ <*NDSS(V,A/S.len S)*> IsNDRankSeq V,A;

- **Theorem 3**: For F being FinSequence st F IsNDRankSeq V,A
  ex S being FinSequence st len S = 1 + len F
  & S IsNDRankSeq V,A &
  for n being Nat st n in dom S holds S.n = NDSS(V,A \ (F.n));

and two simple examples of such sequences:

- **Theorem 4**: <*NDSS(V,A)*> IsNDRankSeq V,A;
- **Theorem 5**: <*NDSS(V,A),NDSS(V,A / NDSS(V,A))*> IsNDRankSeq V,A;

III. OPERATIONS ON NOMINATIVE DATA

We defined the denaming operation on nonatomic nominative data of Type SC for a single name \((v \Rightarrow d)\) as follows:

- **Definition**: For V,A be set
  let v be object;
  let D be NonatomicND of V,A such that v in dom D;
  func denaming(v,D) -> TypeSCNominativeData of V,A equals D.v;
end;

We defined the naming operation on nonatomic nominative data of Type SC for a single name \(\Rightarrow v(d)\) as follows:

- **Definition**: For V,A be set;
  let v,D be object;
  assume D is TypeSCNominativeData of V,A;
  assume v in V;
  func naming(V,A,v,D) -> NonatomicND of V,A equals v .--> D;
end;

We defined the naming operation on nonatomic nominative data of Type SC for a sequence of names \(\Rightarrow v_1v_2...v_n(d)\) as follows:

- **Definition**: For V,A be set;
  let f be V-valued FinSequence;
  func namingSeq(V,A,f,a) -> FinSequence means
  len it = len f &
  t.1 = naming(V,A,f.len D,a) &
  for n being Nat st 1 <= n < len f holds
  it.(n+1) = naming(V,A,f.(len f-n),it.n);
end;

where **V-valued** states that the range of a relation is included in V [44].

Below we state several basic properties of the introduced operations:

- **Theorem 6**: For f being V-valued FinSequence holds
  for n being Nat st 1 <= n < len f holds
  namingSeq(V,A,f,a).n is NonatomicND of V,A;
end;

where \(v, v_1, a, a_1\) are arbitrary objects, \(v .--> a\) is a one element function \([v,a]\) [46], and \((u,v) .--> (a,b)\) is a two element function \([u,a],[v,b]\) [47].
for f being V-valued FinSequence st len f > 0 holds naming(V,A,f,a) is NonatomicND of V,A;

for V being non empty set for v being Element of V holds naming(V,A,<>*v*>,a) = naming(V,A,v,a);

for V being non empty set for v1,v2 being Element of V st for D being TypeSCNominativeData of V,A holds naming(V,A,<>*v1,v2*>,D) = v1.-->(v2.--->D);

The following theorem shows that denaming applied to the result of naming applied to a data d results in d, if denaming and naming concern the same name v:

for D being TypeSCNominativeData of V,A holds v in V implies denaming(v,naming(V,A,v,D)) = D;

The following theorem states an identity for naming applied to the result of denaming:

v in dom D implies naming(V,A,v,denaming(v,D)) = v .--> D.v;

We defined the global overlapping on nominative data of Type SC as follows:

definition
let V,A be set;
let d1,d2 be object such that d1 is TypeSCNominativeData of V,A and d2 is TypeSCNominativeData of V,A;
func global_overlapping(V,A,d1,d2) -> TypeSCNominativeData of V,A means
ex f1,f2 being Function st f1 = d1 & f2 = d2 & it = f2 \/(f1 \ ((dom(f1)) \ (dom(f2))))
if not d1 in A & not d2 in A
otherwise
it = the TypeSCNominativeData of V,A;
end;

We defined the local overlapping with single name parameter v on nominative data of Type SC as follows:

definition
let V,A be set;
let d1,d2,v be object;
func local_overlapping(V,A,d1,d2,v) -> TypeSCNominativeData of V,A equals
global_overlapping(V,A,d1,naming(V,A,v,d2));
end;

Below we state several basic properties of the global overlapping:

for d1,d2 being NonatomicND of V,A st not d1 in A & not d2 in A & dom d1 = dom d2
holds global_overlapping(V,A,d1,d2) = d2;

for V being non empty set for v being Element of V holds naming(V,A,<>*v*>,a) = naming(V,A,v,a);

Below we introduce the set ND(V,A) of all nominative data of Type SC over a set of names V and a set of values A. This is the carrier of the algebra of nominative data.

definition
let V,A be set;
func ND(V,A) -> set equals
the set of all D where D is TypeSCNominativeData of V,A;
end;

Finally, we defined the operations of the algebra of nominative data as functions on ND(V,A) as follows:

definition
let V,A be set;
let v be object;
func denaming(V,A,v) -> PartFunc of ND(V,A),ND(V,A) means
not v.--->a1 in A & not v.--->a2 in A &
a1 is TypeSCNominativeData of V,A &
a2 is TypeSCNominativeData of V,A
implies
global_overlapping(V,A,v.--->a1,v.--->a2) = v.--->a2;

for d1,d2 being NonatomicND of V,A st not d1 in A & not d2 in A & dom d1 = dom d2
holds global_overlapping(V,A,d1,d2) = d2;

for D being TypeSCNominativeData of V,A holds v in V &
not v.--->a1 in A & not v.--->a2 in A &
a1 is TypeSCNominativeData of V,A &
a2 is TypeSCNominativeData of V,A
implies
global_overlapping(V,A,v.--->a1,v.--->a2) = v.--->a2;

ND(V,A) can be also expressed as the union of the range of the function FNDSC(V,A) defined as:

definition
let V,A be set;
func FNDSC(V,A) -> Function means
dom it = NAT & it.0 = A &
for n being Nat holds it.(n+1) = NDSS(V,A \ / it.n);
end;

Finally, we defined the operations of the algebra of nominative data as functions on ND(V,A) as follows:

definition
let V,A be set;
let v be object;
func naming(V,A,v) -> Function means
for D being TypeSCNominativeData of V,A holds it.D = naming(V,A,v,D);
end;

For D being TypeSCNominativeData of V,A equals
global_overlapping(V,A,d1,naming(V,A,v,d2));
end;

Below we state several basic properties of the global overlapping:

for d1,d2 being NonatomicND of V,A st not d1 in A & not d2 in A & dom d1 = dom d2
holds global_overlapping(V,A,d1,d2) = d2;

definition
let V,A be set;
let v be object;
func local_overlapping(V,A,v) -> PartFunc of [[:ND(V,A),ND(V,A):],ND(V,A) equals
dom it = [[: ND(V,A) \ A , ND(V,A) \ A :] &
for \( d_1, d_2 \) being NonatomicND of \( V, A \) at not in \( A \) & not d2 in \( A \) holds it. \( [d_1, d_2] = \) local_overlapping(\( V, A, d_1, d_2, v \));
end;

\[
\begin{align*}
\{ A, B : \} & \text{ is the Cartesian product of sets } A \text{ and } B \ [48].
\end{align*}
\]

IV. CONCLUSION

We have proposed a library of Mizar definitions of the carrier set and the operations of the algebra of nominative data of the type \( TND_{SC} \) (nominative data with simple names and complex values) which are essentially partial variable assignments that allow arbitrarily deep nesting. We have also formalized theorems that describe basic properties of nominative data and operations on them in Mizar. The obtained results can be useful for formalizing semantics of programs that operate in real-time environments and/or process complex data structures and for reasoning about the behavior of such programs. We plan to formalize an extension of the Floyd-Hoare logic [10] in Mizar that allows reasoning about programs by taking advantage of the formalized notion of a nominative data in further papers. We plan to continue the work described in this paper as follows: 1) To introduce notions of predicates and functions on nominative data in Mizar – predicates on nominative data will be used to represent the semantics of pre- and postconditions and functions on nominative data can serve as semantic models of programs. 2) To define operations on partial functions and predicates on nominative data which represent semantics of common programming language constructs such as sequential execution, branching, cycle, etc. Sets of predicates and functions on nominative data together with such operations form a program algebra. 3) To define a special Floyd-Hoare composition using the introduced notions of predicates and functions (programs) on nominative data. 4) To formulate inference rules for the Floyd-Hoare logic for programs on nominative data with partial pre- and postconditions and prove soundness of this inference system in Mizar.

REFERENCES


