






Extended Floyd-Hoare Logic over Relational Nominative Data

Mykola Nikitchenko¹  , Ievgen Ivanov¹,
Artur Kornilowicz² , and Andrii Kryvolap¹

¹ Taras Shevchenko National University of Kyiv,
64/13, Volodymyrska Street, Kyiv 01601, Ukraine
nikitchenko@unicyb.kiev.ua, ivanov.eugen@gmail.com, krivolapa@gmail.com

² Institute of Informatics, University of Białystok,
Ciołkowskiego 1M, 15-245 Białystok, Poland
arturk@math.uwb.edu.pl

Abstract. The classical Floyd-Hoare logic is defined for the case of total pre- and postconditions and partial programs (i.e. programs can be undefined on some input data, but conditions must be defined on all data). In this paper we propose an extension of this logic for the case of partial conditions and partial programs over structured data. These data are based on two constructing primitives: naming and relational structuring and are called relational nominative data. They can conveniently represent many data structures used in programming. The semantics of the proposed logic is represented by special algebras of partial functions and predicates over relational nominative data. Operations of these algebras are called compositions. We present an inference system for the mentioned logic and propose an approach to its formalization in Mizar proof assistant. The obtained results can be used in software verification.

Keywords: Formal methods · Software verification
Floyd-Hoare logic · Proof assistant · Nominative data · Partial table
Relational database

1 Introduction

Floyd-Hoare logic [1–3] is a formal system widely used for reasoning about program correctness. It is based on the notion of a Floyd-Hoare triple (assertion) which consists of a precondition, a program, and a postcondition and means the following requirement: when input data satisfies the precondition, the program output must satisfy the postcondition, if the program terminates. Specification of program properties in terms of Floyd-Hoare triples is natural and reasoning is convenient thanks to a compositional proof system. A survey of the important results on properties of the Hoare’s proof system (soundness, completeness in specific senses) and its extensions was given in [3].

In the classical Floyd-Hoare logic predicates (pre- and postconditions) are assumed to be total (defined on all data) and programs can be partial (in the

sense that if a program does not terminate, its resulting value is undefined). The ability to deal with partiality is an important aspect, because partial operations frequently arise in programming. In most programming languages some basic operations on data such as arithmetic division are already partial. Furthermore, partiality of programs may be caused by non-termination which can arise from loop constructs and/or recursion. For similar reasons partiality can arise in software specifications.

In [4] the following classification of partiality phenomena in software specifications was proposed: *non-termination*, i.e. if evaluation of an expression does not terminate, its value is assumed to be undefined and the operation is considered partial; *error value*, i.e. if some values of an operation's argument are illegal (e.g. division by zero, Pop operation applied to an empty stack, etc.), the result of the operation on such values is assumed to be undefined and the operation is considered partial; and *nondeterminism*, i.e. if a result of an operation on an argument value is not determined uniquely by the specification of this operation (operation is underspecified), the result of application of the operation to such a value is assumed to be undefined and the operation is considered partial. Other opinions on the meanings of partiality in software specifications can be found in [5–7].

In [5] a taxonomy of the ways of dealing with partiality in software specification languages and logics was proposed. Among different approaches notable are excluding partial functions from consideration and providing alternative notations (e.g. graph of a partial function), using a three-valued (many-valued) logic, where the third value represents an undefined result, or making all function applications denote [5]. It should be noted that almost all approaches that try to not allow partial programs and/or predicates that describe program guards or properties explicitly and reduce or translate them to the classical case of total functions and predicates have drawbacks analyzed in detail in [4–6].

A more natural and potentially fruitful approach is to allow partiality in both programs and program specifications and construct non-classical proof systems allowing explicit reasoning about properties of such programs and specifications. This approach is applied in this paper to Floyd-Hoare logic. More specifically, in the classical Floyd-Hoare logic the predicates describing program pre- and postconditions are assumed to be total. But obviously, it is desirable to be able to use partial operations in pre- and postconditions of programs, where partiality may be interpreted in one of the senses proposed in [4]. So it is desirable to obtain an extension of Floyd-Hoare logic that is able to deal with both partial programs and partial predicates.

We consider such an extension in this paper. In the previous works [8,9] we have considered extensions of Floyd-Hoare logic to partial mappings over data represented as partial mappings on named values (called nominative sets) and proposed the corresponding inference systems and investigated their soundness and extensional and intensional completeness. However, nominative sets (which can be considered as partial functions from names to values) naturally represent only a flat data organization in low-level programming. Using Floyd-Hoare logic

with partial mappings over nominative sets for reasoning about programs which operate on complex data structures (e.g. trees) is inconvenient, because one needs to take into account many low-level details about data structure implementation. For this reason, in this paper we propose an extension of Floyd-Hoare logic for the case of partial conditions and partial programs on more general class of data. These data are based on two primitives: hierarchical naming and relational structuring and are called relational nominative data. As was shown in [10] hierarchically nominative data are sufficient for representing many data structures (like multidimensional arrays, lists, trees, etc.) that are frequently used in programming. Relational structuring permits to represent partial tables and relations used in relational databases.

To develop such an extension we will adopt the *composition-nominative approach* [11] to program formalization. This approach aims to propose a mathematical basis for development of formal methods of analysis and synthesis of software systems and is grounded on several principles [12], including the *Development principle* (from abstract to concrete), the *Principle of integrity of intensional and extensional aspects*, the *Principle of priority of semantics over syntax*, *Compositionality principle*, and the *Nominativity principle*. The latter Nominativity principle states that *nominative data* adequately represent various forms of data that are processed and stored in computing systems. Nominative data can be considered as a special class of hierarchically organized data. There exist several types of nominative data [12] (with simple or complex names and with simple or complex values), but all of them are based on naming relations that associate names and values. In the composition-nominative approach 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*. Properties of data are represented as partial predicates on nominative data. Nominative functions and predicates can be composed in many ways, e.g. by sequential composition, branching, and so on. Operations that construct composed systems from constituents are called *compositions*. A set of compositions together with a set of functions obtained from a chosen set of basic functions by applications of compositions forms an algebraic system (*program algebra*) which is a semantic model of a programming language. The syntax of this language follows naturally from this semantic model: programs are represented as terms of the described algebra.

In accordance with the composition-nominative approach the semantic component of our Floyd-Hoare logic extension will be based on program algebra (a set of functions and predicates on nominative data which can be obtained from some chosen basic functions and predicates using a specific set of compositions). In this paper we generalize the notion of nominative data by adding a new constructing primitive that introduces finite relations (sets of nominative data) as name values. Obtained data are called relational nominative data. Such data permit to model relations considered in relational databases. Thus, the carrier sets of our program algebra will consist of partial functions and predicates over relational nominative data with complex names and complex values [10].

We will treat a Floyd-Hoare triple as a composition with two predicates on relational nominative data and a program (a partial binominative function which belongs to the carrier set of the program algebra) as arguments. The predicates represent pre- and postconditions and the result of the composition is a predicate. However, the classical definition of Floyd-Hoare triple validity leads to Floyd-Hoare composition that is not monotone [8]. Monotonicity is one of the key properties used for reasoning about programs. It is also important for reasoning about loop-free programs and using them as approximations of programs with loops. This explains the need of a special definition of Floyd-Hoare composition for the extension of Floyd-Hoare logic on partial predicates which is monotone, but converges to the classical definition, if predicates are total. Such a definition was presented in [8] and we will adapt it to the case considered in this paper.

To make our Floyd-Hoare logic extension practically applicable for program verification one can implement it in a proof assistant software [13].

Many well known proof assistants (e.g. Isabelle, Coq, PVS, etc.) provide a substantial support for reasoning about total functions, programs, predicates and are convenient for either formulating the classical Floyd-Hoare logic axiomatically, or embedding it in their logics. For example, Isabelle proof assistant includes the “Hoare” HOL (Higher-Order Logic)-based theory that provides an implementation of Hoare logic for a simple imperative programming language with WHILE loops following [14,15]. However, a support of reasoning about programs using partial pre- and postconditions is generally not developed.

We propose an approach to formalization of our extended Floyd-Hoare logic which supports partial pre- and postconditions in the proof assistant Mizar [16,17]. This proof assistant is based on first-order logic and axiomatic set theory (Tarski-Grothendieck set theory [18]). The Mizar system has its own proof verifier¹ used to verify the logical correctness of proofs written in the Mizar language – a declarative language designed to write mathematical documents. It contains rules for writing traditional mathematical items (e.g. definitions, theorems, proof steps, etc.) and also provides syntactic constructions to launch specialized procedures (e.g. term identifications, term reductions [22], flexary connectives [23], definitional expansions [24]) which increase the computational power of the verifier (e.g. equational calculus [25,26], processing properties of functors and predicates [27–29]). An important component of the Mizar system is its library of formalized mathematical theories called Mizar Mathematical Library (MML). It 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.² Consequently, Mizar has well developed tools for working with partial functions and predicates and is well-suited for our purposes. Besides, the Mizar system has a degree of proof

¹ Research on using specialized external systems to increase computational power of the Mizar system is also conducted [19–21].

² Due to the size, the MML is a subject of research on optimization of theorems and definitions [33]. It includes the improvement of legibility of proofs [34–36] and removing duplications.

automation support such as discovery of a list of proven facts that imply the current goal which may be used as basis for implementing software verification in a semi-automatic mode.

To simplify and partially automate application of Floyd-Hoare logic to proving program properties it is convenient to have a corresponding system of inference rules. The traditional inference system for the language WHILE [37] is sound and extensionally complete for the classical Floyd-Hoare logic with total predicates [37] (extensional completeness means that pre- and postconditions may be arbitrary predicates; intensional completeness means that pre- and postconditions should be presented by formulas of a given language). Soundness and completeness are important for practical applicability of an inference system (if a system is not sound, assertions that can be inferred using this system may be false; if a system is not complete, some of the valid assertions could be impossible to infer). However, this inference system is not sound and complete for partial predicates as was shown in [8].

To deal with the soundness and completeness problems we will modify the traditional inference system for the language WHILE and introduce additional constraints on inference rules that correspond to the new definition of validity of Floyd-Hoare assertions, and investigate its soundness and extensional completeness. The obtained results extend the results concerning inference systems for Floyd-Hoare logic with partial predicates over flat (non-hierarchical) data obtained in [8,9].

The paper is organized in the following way. In Sect. 2 we describe the notion of relational nominative data and define main operations on them. In Sect. 3 we describe our semantics based on Floyd-Hoare logic. In Sect. 4 we specify the syntax of our extended Floyd-Hoare logic. In Sect. 5 we propose an inference system for our logic and consider problems of its soundness and completeness. In Sect. 6 we describe an approach to formalization of our extended Floyd-Hoare logic in Mizar. In Sect. 7 we describe the related work. In Sect. 8 we give conclusions.

2 Algebra of Relational Nominative Data

In the composition-nominative approach data are treated as nominative data. There are several types of nominative data, but all of them are based on naming relations. The simplest type of nominative data is the class of *nominative sets* which are partial mappings from a set of names (program variables) to a set of basic values. Other types of nominative data represent hierarchical data organizations [10]. Here we present the definition of relational nominative data. Before giving such definitions, let us introduce the following notation.

To distinguish total functions from partial we will use the symbol \xrightarrow{p} for partial functions and \xrightarrow{t} for total. We will also use the symbol \xrightarrow{n} for partial functions with finite graphs. For any partial function $f : D \xrightarrow{p} D'$ on some set D :

- $f(d) \downarrow$ denotes that f is defined on $d \in D$;
- $f(d) \downarrow = d'$ denotes that f is defined on $d \in D$ with a value $d' \in D'$;
- $f(d) \uparrow$ denotes that f is undefined on $d \in D$;
- $\text{dom}(f) = \{d \in D \mid f(d) \downarrow\}$ is the domain of a function (note that in different branches of mathematics there exist different definitions of the domain of a partial function; we will adopt the convention used in recursion theory).

We will denote by $f_1(d_1) \cong f_2(d_2)$ the *strong equality*, i.e. the condition that $f_1(d_1) \downarrow$ if and only if $f_2(d_2) \downarrow$, and if $f_1(d_1) \downarrow$, then $f_1(d_1) = f_2(d_2)$.

For any nonempty set V we will denote by V^+ the set of all nonempty finite sequences (*words*) of elements of V . For any word $u \in V^+$ we will denote by $|u|$ its length. If $u, v \in V^+$, we will denote by uv the concatenation of u and v . We will write $u \leq v$, if u is a prefix of v , and $u < v$, if $u \leq v$, $u \neq v$.

For any set of names V and a set of basic values (atoms) A the corresponding class ${}^V A$ of *nominative sets* is defined as

$${}^V A = V \xrightarrow{p} A.$$

We chose V to denote the set of names because we are oriented on mathematical logic where V is practically standard notation for a set of variables (names). We will use the following notations for nominative sets:

- $[v_1 \mapsto a_1, \dots, v_n \mapsto a_n]$, where v_1, \dots, v_n are names from V and a_1, \dots, a_n are atoms from A , denotes a nominative set with the graph $\{(v_1, a_1), \dots, (v_n, a_n)\}$;
- $[v_i \mapsto a_i \mid i \in I]$, where I is some set of indices, means a nominative set with the graph $\{(v_i, a_i) \mid i \in I\}$;
- $v \mapsto a \in d$, where d is a nominative set, means that $d(v) \downarrow = a$, i.e. the value of the variable v in d is a ;
- $[\]$ denotes the empty nominative set (a nowhere defined function).

Relational nominative data are built over classes of names V and basic values A using a naming construction of the form $[v_1 \mapsto d_1, \dots, v_n \mapsto d_n]$, and a relational construction of the form $\{d_1, \dots, d_n\}$ where v_1, \dots, v_n are different names from V and d_1, \dots, d_n are either atoms or other relational nominative data.

Relational nominative data are classified in accordance with the following parameters [10]: *names* can be simple (unstructured) or complex (structured), *values* can be simple (unstructured) or complex (structured). Within the class of complex structured values we distinguish the class of relational values of the form $\{d_1, \dots, d_n\}$. To define the notion of a complex name we will use the Development principle (from abstract to concrete) and consider the simplest case of name construction: complex names are *sequences* of simple names which satisfy the associativity property [10]. More specifically, we will assume that complex names are constructed with the help of concatenation operation (which is associative). We will adopt the following *Principle of associative construction and processing of complex names* [10]: complex names are constructed from simple names using

concatenation, and data with complex names must be processed by operations that take into account associativity of names. Moreover, we will require that data with complex names satisfy the *Principle of unambiguous associative naming* [10]: one complex name must have at most one corresponding value in any given data.

Let us give the formal definition of the class $RND(V, A)$ of relational nominative data with complex names and complex values. We will assume that V and A are fixed nonempty sets of *simple names* and *basic values*. We will call the elements of V^+ *complex names*.

First, we define

$$RNDs(V, A) = \bigcup_{k \geq 0} RNDs_k(V, A),$$

where

$$RNDs_0(V, A) = A \cup \{\emptyset\},$$

$$RNDs_{k+1}(V, A) = RNDs_k(V, A) \cup (V^+ \xrightarrow{n} RNDs_k(V, A)) \cup R(RNDs_k(V, A)).$$

Here \xrightarrow{n} denotes a constructor of nominative sets and R denotes a construction of finite relations:

$$R(X) = \{Y \subseteq X \mid Y \text{ is finite}\}.$$

The class $RNDs(V, A)$ uses complex names in its construction, but possible ambiguity of naming is not taken into consideration. Thus, we add additional restrictions to obtain the class $RND(V, A)$.

Naming structure can be represented by oriented trees with arcs labeled by names and leafs labeled by atoms, empty nominative set, or relations. We will call any finite sequence of names $p = (v_1, v_2, \dots, v_k)$ a *path*. A *path in a given data* $d \in RNDs(V, A)$ is a path (v_1, v_2, \dots, v_k) such that the value of the expression $(\dots((d(v_1))(v_2))\dots(v_k))$ is defined (it corresponds to a path from the root to some vertex in a tree). If $pt = (v_1, v_2, \dots, v_k)$ is a path in d , we will say that $(\dots((d(v_1))(v_2))\dots(v_k))$ is the value of pt in d and denote it as $d(v_1, v_2, \dots, v_k)$.

A *terminal path* is a path with atomic, empty value or relational value.

Data of the class $RND(V, A)$ are elements of the set $RNDs(V, A)$ such that for any d and any two paths (u_1, u_2, \dots, u_k) and (v_1, v_2, \dots, v_l) in d , neither of which is a prefix of another, words $u_1 u_2 \dots u_k$ and $v_1 v_2 \dots v_l$ are incomparable in the sense of prefix relation (*principle of unambiguous associative naming*). This principle should be applied to all subdata within d .

In [10] it was shown how conventional data structures can be represented by different kinds of relational nominative data.

Let $Nd(V, A) = (V^+ \xrightarrow{n} RND(V, A)) \cap RND(V, A)$ and

$$Rd(V, A) = R(RND(V, A))$$

be the subclasses of $RND(V, A)$ called classes of nominative and relational data respectively.

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

The *nominative rank* of $d \in RND(V, A)$ is the greatest length of terminal paths in d . For any word $u \in V^+$ and any data $d \in RND(V, A)$ let us denote

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

(*division* of d by u).

Definition 1. *Associative denaming*

$$v \Rightarrow_a: RND(V, A) \xrightarrow{p} RND(V, A)$$

is an operation with a parameter $v \in V^+$. On $Rd(V, A)$ it is undefined, and on $Nd(V, A)$ is defined by induction on the length of v as follows:

- if $|v| = 1$, then $v \Rightarrow_a (d) = d(v)$, if $d(v) \downarrow$; $v \Rightarrow_a (d) = d/v$ if $d(v) \uparrow$ and $d/v \neq []$, and $v \Rightarrow_a (d) \uparrow$ otherwise.
- if $|v| = n > 1$, then $v \Rightarrow_a (d) \cong v_1 \Rightarrow_a (x \Rightarrow_a (d))$, where $v = xv_1$, $x \in V$, $v_1 \in V^{n-1}$ (*principle of associative denaming*).

For example,

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

It is easy to check that $v \Rightarrow_a$ satisfies the following property (associativity)

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

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 is an operation*

$$\Rightarrow v: RND(V, A) \xrightarrow{t} RND(V, A)$$

with a parameter $v \in V^+$ such that

$$\Rightarrow v(d) = [v \mapsto d].$$

Overlapping can be considered as an operation which updates values in the first argument with the values from the second argument. It joins two data and resolves name conflicts in favor of its second argument. We will define two kinds of overlapping: global and local. Global overlapping can be used for formalization of procedure calls and the local overlapping formalizes the assignment operator in programming languages.

Definition 3. *Global overlapping is a binary operation*

$$\nabla_a : RND(V, A) \times RND(V, A) \xrightarrow{P} RND(V, A)$$

defined inductively by the nominative rank of the first argument as follows.

Let $Nd_k(V, A)$ be the class of data with the nominative rank not greater than k .

Induction base. If $d_1 \in Nd_0(V, A)$, then

$$d_1 \nabla_a d_2 \cong \begin{cases} d_2, & d_1 = [] \text{ and } d_2 \in Nd(V, A) \setminus A; \\ \text{undefined,} & \text{in other cases} \end{cases}$$

Induction step. Assume that the value $d_1 \nabla_a d_2$ is defined for all d_1, d_2 such that $d_1 \in Nd_k(V, A)$. Let $d_1 \in Nd_{k+1}(V, A)$ and $d_1 \notin Nd_k(V, A)$. Then $d_1 \nabla_a d_2 = d$, where $d \in Nd(V, A)$ is defined by its values on names $u \in V^+$:

- $d(u) = d_2(u)$, if $u \in \text{dom}(d_2)$ and u does not have a proper prefix which belongs to $\text{dom}(d_1)$;
- $d(u) = d_1(u) \nabla_a (d_2/u)$, if $d_1(u) \downarrow$ and $d_1(u) \notin A$, and u is a proper prefix of some element of $\text{dom}(d_2)$;
- $d(u) = d_2/u$, if $d_1(u) \downarrow$ and $d_1(u) \in A$, and u is a proper prefix of some element of $\text{dom}(d_2)$;
- $d(u) = d_1(u)$, if $d_1(u) \downarrow$ and u is not comparable (in the sense of prefix relation) with any element of $\text{dom}(d_2)$;
- $d(u) \uparrow$, otherwise.

The following examples illustrate this operation:

1. $[u \mapsto d_1] \nabla_a [v \mapsto d_2] = [u \mapsto d_1, v \mapsto d_2]$, if u, v are incomparable in the sense of prefix relation;
2. $[uv \mapsto d_1] \nabla_a [u \mapsto d_2] = [u \mapsto d_2]$, i.e. a value of a name in the second argument overwrites the value of extension of this name in the first argument;
3. $[u \mapsto d_1] \nabla_a [uv \mapsto d_2] = [u \mapsto (d_1 \nabla_a [v \mapsto d_2])]$ ($d_1 \notin A$), i.e., a value of a name in the second argument modifies values of prefixes of this name in the first argument.

Definition 4. *Local overlapping is an operation*

$$\nabla_a^v : RND(V, A) \xrightarrow{P} RND(V, A)$$

with a parameter $v \in V^+$ such that

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

For example, $[u \mapsto 1] \nabla_a^v [w \mapsto 2] = [u \mapsto 1, v \mapsto [w \mapsto 2]]$.

Now we define operations that operate also on relational data. Therefore we consider two cases: data is a nominative data and data is a relational data. Nominative data are often considered as sets.

Definition 5. *Union \cup is an operation*

$$\cup : RND(V, A) \times RND(V, A) \xrightarrow{p} RND(V, A)$$

defined for any $d_1, d_2 \in RND(V, A)$ in the following way:

- $d_1 \cup d_2 = [v \mapsto d' | v \mapsto d' \in d_1 \text{ or } v \mapsto d' \in d_2]$, if $d_1, d_2 \in Nd(V, A)$ and names from $\text{dom}(d_1)$ and $\text{dom}(d_2)$ are pairwise incomparable;
- $d_1 \cup d_2 = \{d' | d' \in d_1 \text{ or } d' \in d_2\}$, if $d_1, d_2 \in Rd(V, A)$;
- undefined in other cases.

An example of the union of elements of $Nd(V, A)$: if

$$d_1 = [x_1 \mapsto 1, x_2 \mapsto 2], \quad d_2 = [x_3 \mapsto 3, x_4 \mapsto 4],$$

then $d_1 \cup d_2 = [x_1 \mapsto 1, x_2 \mapsto 2, x_3 \mapsto 3, x_4 \mapsto 4]$.

An example of the union of elements of $Rd(V, A)$: if

$$d_1 = \{[u \mapsto 1, v \mapsto 2], [u \mapsto 3, v \mapsto 4]\},$$

$$d_2 = \{[u \mapsto 5, v \mapsto 6], [u \mapsto 7, v \mapsto 8]\},$$

then $d_1 \cup d_2 = \{[u \mapsto 1, v \mapsto 2], [u \mapsto 3, v \mapsto 4], [u \mapsto 5, v \mapsto 6], [u \mapsto 7, v \mapsto 8]\}$.

Definition 6. *Difference \setminus is an operation*

$$\setminus : RND(V, A) \times RND(V, A) \xrightarrow{p} RND(V, A)$$

defined for any $d_1, d_2 \in RND(V, A)$ in the following way:

- $d_1 \setminus d_2 = [v \mapsto d' | v \mapsto d' \in d_1 \text{ and } v \mapsto d' \notin d_2]$, if $d_1, d_2 \in Nd(V, A)$;
- $d_1 \setminus d_2 = \{d' | d' \in d_1 \text{ and } d' \notin d_2\}$, if $d_1, d_2 \in Rd(V, A)$;
- undefined in other cases.

An example of the difference of elements of $Nd(V, A)$: if

$$d_1 = [x_1 \mapsto 1, x_2 \mapsto 2], \quad d_2 = [x_1 \mapsto 1, x_4 \mapsto 4],$$

then $d_1 \setminus d_2 = [x_2 \mapsto 2]$.

An example of the difference of elements of $Rd(V, A)$: if

$$d_1 = \{[u \mapsto 1, v \mapsto 2], [u \mapsto 3, v \mapsto 4]\},$$

$$d_2 = \{[u \mapsto 1, v \mapsto 2], [u \mapsto 5, v \mapsto 6]\},$$

then $d_1 \setminus d_2 = \{[u \mapsto 3, v \mapsto 4]\}$.

Definition 7. *Product \otimes is an operation*

$$\otimes : RND(V, A) \times RND(V, A) \xrightarrow{p} RND(V, A)$$

defined for any $d_1, d_2 \in RND(V, A)$ in the following way:

- $d_1 \otimes d_2 = \{d'_1 \cup d'_2 \mid d'_1 \in d_1, d'_2 \in d_2\}$, if $d_1, d_2 \in Rd(V, A)$;
- *undefined in other cases.*

An example of the product of elements of $Rd(V, A)$: if

$$d_1 = \{[u \mapsto 1, v \mapsto 2], [u \mapsto 3, v \mapsto 4]\}, d_2 = \{[w \mapsto 3]\},$$

then $d_1 \otimes d_2 = \{[u \mapsto 1, v \mapsto 2, w \mapsto 3], [u \mapsto 3, v \mapsto 4, w \mapsto 3]\}$.

Definition 8. *Projection π^{v_1, \dots, v_n} is an operation*

$$\pi^{v_1, \dots, v_n} : RND(V, A) \xrightarrow{p} RND(V, A)$$

with parameters $v_1, \dots, v_n \in V^+$ such that v_1, \dots, v_n are pairwise incomparable names, defined for any $d \in RND(V, A)$ in the following way:

- $\pi^{v_1, \dots, v_n}(d) = [v \mapsto d(v) \mid v \in \{v_1, \dots, v_n\}, v \in \text{dom}(d)]$, if $d \in Nd(V, A)$;
- $\pi^{v_1, \dots, v_n}(d) = \{\pi^{v_1, \dots, v_n}(d') \mid d' \in d, d' \in Nd(V, A)\}$, if $d \in Rd(V, A)$;
- *undefined in other cases.*

An example of the projection of elements of $Nd(V, A)$: if

$$d = [x_1 \mapsto [a \mapsto 1, b \mapsto 2], x_2 \mapsto [a \mapsto 1, c \mapsto 2], x_3 \mapsto []],$$

then $\pi^{x_1, x_3}(d) = [x_1 \mapsto [a \mapsto 1, b \mapsto 2], x_3 \mapsto []]$.

An example of the projection of elements of $Rd(V, A)$: if

$$d = \{[u \mapsto 1, v \mapsto 2], [u \mapsto 3, v \mapsto 4]\},$$

then $\pi^v(d) = \{[v \mapsto 2], [v \mapsto 4]\}$.

Definition 9. *Deleting δ^{v_1, \dots, v_n} is an operation*

$$\delta^{v_1, \dots, v_n} : RND(V, A) \xrightarrow{p} RND(V, A)$$

with parameters $v_1, \dots, v_n \in V^+$ defined for any $d \in RND(V, A)$ in the following way:

- $\delta^{v_1, \dots, v_n}(d) = [u \mapsto d(u) \mid u \in \text{dom}(d), u \notin \{v_1, \dots, v_n\}]$, if $d \in Nd(V, A)$;
- $\delta^{v_1, \dots, v_n}(d) = \{\delta^{v_1, \dots, v_n}(d') \mid d' \in d, d' \in Rd(V, A)\}$, if $d \in Rd(V, A)$;
- *undefined in other cases.*

An example of an application of deletion to elements of $Nd(V, A)$: if

$$d = [x_1 \mapsto [a \mapsto 1, b \mapsto 2], x_2 \mapsto [a \mapsto 1, c \mapsto 2], x_3 \mapsto []],$$

then $\delta^{x_2}(d) = [x_1 \mapsto [a \mapsto 1, b \mapsto 2], x_3 \mapsto []]$.

An example of an application of deletion to elements of $Rd(V, A)$: if

$$d = \{[u \mapsto 1, v \mapsto 2], [u \mapsto 3, v \mapsto 4]\},$$

then $\delta^u(d) = \{[v \mapsto 2], [v \mapsto 4]\}$.

Definition 10. Renaming $r_{u_1, \dots, u_n}^{v_1, \dots, v_n}$ is an operation

$$r_{u_1, \dots, u_n}^{v_1, \dots, v_n} : RND(V, A) \xrightarrow{P} RND(V, A)$$

with parameters $v_1, \dots, v_n, u_1, \dots, u_n \in V^+$ such that v_1, \dots, v_n are pairwise incomparable names, defined for any $d \in RND(V, A)$ in the following way:

- $r_{u_1, \dots, u_n}^{v_1, \dots, v_n}(d) = \delta^{v_1, \dots, v_n}(d) \cup [v \mapsto d(u) \mid u \in \text{dom}(d)]$, if $d \in Nd(V, A)$;
- $r_{u_1, \dots, u_n}^{v_1, \dots, v_n}(d) = \{r_{u_1, \dots, u_n}^{v_1, \dots, v_n}(d') \mid d' \in d, d' \in Rd(V, A)\}$, if $d \in Rd(V, A)$;
- undefined in other cases.

An example of application of renaming to elements of $Nd(V, A)$: if

$$d = [x_1 \mapsto [a \mapsto 1], x_2 \mapsto [a \mapsto 1, b \mapsto 2]],$$

then $r_{x_1, x_2}^{y_1, x_1}(d) = [x_2 \mapsto [a \mapsto 1, b \mapsto 2], y_1 \mapsto [a \mapsto 1], x_1 \mapsto [a \mapsto 1, b \mapsto 2]]$.

An example of application of renaming to elements of $Rd(V, A)$: if

$$d = \{[v \mapsto 2], [v \mapsto 4]\},$$

then $r_v^u(d) = \{[v \mapsto 2, u \mapsto 2], [v \mapsto 4, u \mapsto 4]\}$.

Definition 11. Natural join \bowtie is an operation

$$\bowtie : RND(V, A) \times RND(V, A) \xrightarrow{P} RND(V, A)$$

defined for any $d_1, d_2 \in RND(V, A)$ in the following way:

- $d_1 \bowtie d_2 = d_1 \cup d_2$,
if $d_1, d_2 \in Nd(V, A)$ and $d_1(v) = d_2(v)$ for any $v \in \text{dom}(d_1) \cap \text{dom}(d_2)$;
- $d_1 \bowtie d_2 = \{d'_1 \bowtie d'_2 \mid d'_1 \in d_1, d'_2 \in d_2, d'_1 \bowtie d'_2 \text{ is defined}\}$,
if $d_1, d_2 \in Rd(V, A)$;
- undefined in other cases.

An example of the natural join of elements of $Nd(V, A)$: if

$$d_1 = [x \mapsto 1, y \mapsto 2], \quad d_2 = [y \mapsto 2, z \mapsto 3],$$

then $d_1 \bowtie d_2 = [x \mapsto 1, y \mapsto 2, z \mapsto 3]$.

An example of the natural join of elements of $Rd(V, A)$: if

$$d_1 = \{[u \mapsto 1, v \mapsto 2], [u \mapsto 3, v \mapsto 4]\},$$

$$d_2 = \{[v \mapsto 2, w \mapsto 3]\},$$

then $d_1 \bowtie d_2 = \{[u \mapsto 1, v \mapsto 2, w \mapsto 3]\}$.

Definition 12. Division \div is an operation

$$\div : RND(V, A) \times RND(V, A) \xrightarrow{P} RND(V, A)$$

defined for any $d_1, d_2 \in RND(V, A)$ as:

$$d_1 \div d_2 \cong \bigcup \{d \in RND(V, A) \mid d \otimes d_2 \subseteq d_1\}.$$

An example of the division of elements of $Rd(V, A)$: if

$$d_1 = \{[u \mapsto 1, v \mapsto 1], [u \mapsto 1, v \mapsto 2], [u \mapsto 2, v \mapsto 1]\},$$

$$d_2 = \{[v \mapsto 2]\},$$

then $d_1 \div d_2 = \{[u \mapsto 1]\}$.

Definition 13. *An algebra of relational nominative data $RNDA(V, A)$ is an algebra with the carrier $RND(V, A)$ and the operations*

$$\Rightarrow, v, v \Rightarrow_a, \nabla_a^v, \cup, \setminus, \otimes, \pi^{v_1, \dots, v_n}, \delta^{v_1, \dots, v_n}, \tau_{u_1, \dots, u_n}^{v_1, \dots, v_n}, \bowtie, \div.$$

3 Semantics of Extended Floyd-Hoare Logic

We treat programs as being defined over nominative data.

Let $Bool = \{F, T\}$ be the set of Boolean values, $Pr^{V, A} = RND(V, A) \xrightarrow{p} Bool$ be the set of *partial predicates*. They can be used to represent semantics of conditions in programs.

Let $FPr^{V, A} = Nd(V, A) \xrightarrow{p} Nd(V, A)$. The elements of $FPr^{V, A}$ are called *binominative functions*. They can be used to represent semantics of programs.

Multi-sorted algebras on sets of partial predicates and partial binominative functions can be used to define semantics of program logics [11, 38]. The operations of such algebras will be called *compositions*.

There are many possible ways to define compositions that provide means to construct complex programs from simpler ones. We have chosen the following compositions to include them as basic to the logics of program level (level of binominative functions):

- parametric assignment composition AS^x which corresponds to assignment operator $:=$;
- composition of identical program id which corresponds to the *skip* operator of the WHILE language;
- composition of sequential execution \bullet ;
- conditional composition IF which corresponds to the *if-then-else* operator;
- cycle (loop) composition WH which corresponds to the *while-do* operator;
- superpositions S_F^x which correspond to procedure calls.

We also need compositions that provide the possibility to construct different kinds of expressions (functions) and conditions (predicates) that are program components. Thus, we will include into the list of compositions superposition compositions.

Finally, to construct predicates describing properties of programs we define the Floyd-Hoare composition FH . It takes a precondition, a postcondition, and a program as inputs and yields a predicate that represents respective Floyd-Hoare assertion. We will also define a composition of *preimage predicate transformer* inspired by weakest precondition introduced by Dijkstra [39].

Let us give definitions of the mentioned compositions.

Definition 14. *Disjunction is a binary composition*

$$\vee : Pr^{V,A} \times Pr^{V,A} \xrightarrow{t} Pr^{V,A}$$

such that for all $p, q \in Pr^{V,A}$ and $d \in RND(V, A)$:

$$(p \vee q)(d) = \begin{cases} T, & \text{if } p(d) \downarrow = T \text{ or } q(d) \downarrow = T, \\ F, & \text{if } p(d) \downarrow = F \text{ and } q(d) \downarrow = F, \\ \text{undefined} & \text{in other cases.} \end{cases}$$

Definition 15. *Negation is a unary composition*

$$\neg : Pr^{V,A} \xrightarrow{t} Pr^{V,A}$$

such that for all $p \in Pr^{V,A}$ and $d \in RND(V, A)$:

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

We will consider *conjunction* $p \wedge q$ of predicates p, q as an abbreviation for $\neg(\neg p \vee \neg q)$.

Definition 16. *Existential quantification over hierarchical data is a unary composition*

$$\exists x : Pr^{V,A} \xrightarrow{t} Pr^{V,A}$$

with a parameter $x \in V^+$ such that for all $p \in Pr^{V,A}$ and $d \in RND(V, A)$:

$$(\exists x p)(d) = \begin{cases} T, & \text{if } p(d \nabla_a^x d') \downarrow = T \text{ for some } d' \in RND(V, A), \\ F, & \text{if } p(d \nabla_a^x d') \downarrow = F \text{ for all } d' \in RND(V, A), \\ \text{undefined} & \text{in other cases.} \end{cases}$$

For each $n = 1, 2, 3, \dots$ denote by $\bar{U}_n(V)$ the set of all tuples $(x_1, \dots, x_n) \in (V^+)^n$ of n complex names such that x_1, x_2, \dots, x_n are pairwise incomparable in the sense of prefix relation \leq .

Also, let us denote $\bar{U}(V) = \bigcup_{n=1}^{\infty} \bar{U}_n(V)$.

Definition 17. *For each $n = 1, 2, 3, \dots$, superposition of n functions into a function is a $n+1$ -ary composition*

$$S_F^{\bar{x}} : (FPr g^{V,A})^{n+1} \xrightarrow{t} FPr g^{V,A}$$

with a parameter $\bar{x} = (x_1, \dots, x_n) \in \bar{U}_n(V)$ such that for all $f, g_1, \dots, g_n \in FPr g^{V,A}$ and $d \in RND(V, A)$:

$$S_F^{\bar{x}}(f, g_1, \dots, g_n)(d) \cong f(d \nabla_a [x_1 \mapsto g_1(d), \dots, x_n \mapsto g_n(d)]).$$

Definition 18. For each $n = 1, 2, 3, \dots$, superposition of n functions into a predicate is a $n+1$ -ary composition

$$S_P^{\bar{x}} : Pr^{V,A} \times (FPr_g^{V,A})^n \xrightarrow{t} Pr^{V,A}$$

with a parameter $\bar{x} = (x_1, \dots, x_n) \in \bar{U}_n(V)$ such that for all $p \in Pr^{V,A}$, $g_1, \dots, g_n \in FPr_g^{V,A}$, and $d \in RND(V, A)$:

$$S_P^{\bar{x}}(p, g_1, \dots, g_n)(d) \cong p(d \nabla_a [x_1 \mapsto g_1(d), \dots, x_n \mapsto g_n(d)]).$$

Definition 19. Denomination is a null-ary composition $'x : FPr_g^{V,A}$ with a parameter $x \in V^+$ such that for each $d \in RND(V, A)$:

$$'x(d) \cong x \Rightarrow_a (d).$$

Definition 20. Assignment over hierarchical data is a composition

$$AS^x : FPr_g^{V,A} \xrightarrow{t} FPr_g^{V,A}$$

with a parameter $x \in V^+$ such that for each $f \in FPr_g^{V,A}$ and $d \in RND(V, A)$:

$$AS^x(f)(d) \cong d \nabla_a^x f(d).$$

Definition 21. Identity program composition is a null-ary composition $id : FPr_g^{V,A}$ such that for each $d \in RND(V, A)$:

$$id(d) = d.$$

Definition 22. Sequential execution is a binary composition

$$\bullet : FPr_g^{V,A} \times FPr_g^{V,A} \xrightarrow{t} FPr_g^{V,A}$$

such that for all $f, g \in FPr_g^{V,A}$ and $d \in RND(V, A)$:

$$(f \bullet g)(d) \cong g(f(d)).$$

Definition 23. Branching is a ternary composition

$$IF : Pr^{V,A} \times FPr_g^{V,A} \times FPr_g^{V,A} \xrightarrow{t} FPr_g^{V,A}$$

such that for all $r \in Pr^{V,A}$ (condition), $f, g \in FPr_g^{V,A}$ (branches bodies), and $d \in RND(V, A)$:

$$IF(r, f, g)(d) = \begin{cases} f(d), & \text{if } r(d) \downarrow = T \text{ and } f(d) \downarrow, \\ g(d), & \text{if } r(d) \downarrow = F \text{ and } g(d) \downarrow, \\ \text{undefined} & \text{in other cases.} \end{cases}$$

Definition 24. *While cycle is a binary composition*

$$WH : Pr^{V,A} \times FPr g^{V,A} \xrightarrow{t} FPr g^{V,A}$$

such that for each $p \in Pr^{V,A}$ (condition), $f \in FPr g^{V,A}$ (loop body), and $d \in RND(V, A)$:

$$WH(p, f)(d) \downarrow = f^{(n)}(d),$$

if $n \geq 0$ such that $p(f^{(i)}(d)) \downarrow = T$ for all $i = 0, 1, \dots, n-1$ and $p(f^{(n)}(d)) \downarrow = F$, where $f^{(i)}$ denotes $\underbrace{f \bullet f \bullet \dots \bullet f}_i$ and $f^{(0)} = id$; and $WH(p, f)(d) \uparrow$, otherwise.

Definition 25. *Monotone Floyd-Hoare composition*

$$FH : Pr^{V,A} \times FPr g^{V,A} \times Pr^{V,A} \xrightarrow{t} Pr^{V,A}$$

is a composition such that for all $p, q \in Pr^{V,A}$ (pre- and postcondition), $f \in FPr g^{V,A}$ (program), and $d \in RND(V, A)$:

$$FH(p, f, q)(d) = \begin{cases} T, & \text{if } p(d) \downarrow = F \text{ or } q(f(d)) \downarrow = T, \\ F, & \text{if } p(d) \downarrow = T \text{ and } q(f(d)) \downarrow = F, \\ \text{undefined} & \text{in other cases.} \end{cases}$$

Definition 26. *Predicate transformer composition is a binary composition*

$$PC : FPr g^{V,A} \times Pr^{V,A} \xrightarrow{t} Pr^{V,A}$$

such that for all $q \in Pr^{V,A}$, $f \in FPr g^{V,A}$, and $d \in RND(V, A)$:

$$PC(f, q)(d) = \begin{cases} T, & \text{if } f(d) \downarrow \text{ and } q(f(d)) \downarrow = T, \\ F, & \text{if } f(d) \downarrow \text{ and } q(f(d)) \downarrow = F, \\ \text{undefined} & \text{in other cases.} \end{cases}$$

Predicate transformer composition is the same as *Glushkov prediction operation* (sequential execution of a function and a predicate). We call this composition (defined for partial predicates) as preimage predicate transformer composition in order to relate it to the *weakest precondition* predicate transformer [39]. Note that $FH(p, pr, q) = p \rightarrow PC(pr, q)$.

The Floyd-Hoare composition is called monotone, because it satisfies the following property, as was shown in [8]:

$$p \subseteq p', q \subseteq q', f \subseteq f' \Rightarrow FH(p, f, q) \subseteq FH(p', f', q'),$$

where inclusion \subseteq is understood as inclusion of the graphs of functions and predicates.

We also need to raise the level of $RNDA(V, A)$ operations to the level of compositions. It means, that operations $\Rightarrow v$, $v \Rightarrow a$ we treat as nullary compositions of the type $FPr g^{V,A}$, operations \cup , \setminus , \bowtie as binary compositions of the type $FPr g^{V,A} \times FPr g^{V,A} \xrightarrow{t} FPr g^{V,A}$, and π^{v_1, \dots, v_n} , δ^{v_1, \dots, v_n} , $r_{u_1, \dots, u_n}^{v_1, \dots, v_n}$ as unary compositions of the type $FPr g^{V,A} \xrightarrow{t} FPr g^{V,A}$. We will use the same symbols both for data algebra operations and compositions.

Definition 27. A relational nominative program algebra $RNPA(V, A)$ is a two-sorted algebra $\langle Pr^{V,A}, FPr^{V,A}; \vee, \neg, \exists x, S_P^{\bar{x}}, S_F^{\bar{x}}, 'x, id, AS^x, \bullet, IF, WH, FH, PC, \Rightarrow v, v \Rightarrow_a, \nabla_a^v, \cup, \setminus, \otimes, \pi^{v_1, \dots, v_n}, \delta^{v_1, \dots, v_n}, r_{u_1, \dots, u_n}^{v_1, \dots, v_n}, \bowtie, \div \rangle$.

This algebra is the semantic base of our extension of Floyd-Hoare logic to partial predicates and (hierarchical) nominative data with complex names and complex values.

4 Syntax and Interpretation

Algebra $RNPA(V, A)$ has strong expressive power that is not required for our goal: to construct a special program logic. Therefore we restrict syntactically the class of terms of this algebra. The idea is to consider programs as binominative functions constructed with the help of compositions $id, AS^x, \bullet, IF, WH, S_F^{\bar{x}}$. Functions of other types (including relational data) can be represented by functional expressions. Formulas represent nominative predicates. The signature of the constructed logic is $\Sigma = (V, Ps, FEs, Prgs)$ where $Ps, FEs, Prgs$ are sets of predicates, functions, and program symbols.

Let us give definitions of the sets of program texts Pt^Σ , formulas Fr^Σ , functional expressions FE^Σ , and Floyd-Hoare assertions $FHFr^\Sigma$.

The sets Pt^Σ, FE^Σ , and Fr^Σ are defined inductively (here we use the symbols of compositions in the purely syntactic sense, i.e. they are currently not associated with semantics, also, for parameters from V^+ respective restrictions hold):

1. if $prs \in Prs$, then $prs \in Fr^\Sigma$;
2. if $fes \in FEs$, then $fes \in FE^\Sigma$;
3. if $prgs \in Prgs$, then $prgs \in Pt^\Sigma$;
4. if $\Phi, \Psi \in Fr^\Sigma$, then $\Phi \vee \Psi, \neg\Phi, \exists x \in Fr^\Sigma$;
5. if $fe, fe_1, fe_2 \in FEs$, then $\Rightarrow v, v \Rightarrow_a, 'x, fe_1 \nabla_a^v fe_2, fe_1 \cup fe_2, fe_1 \setminus fe_2, \otimes, \pi^{v_1, \dots, v_n}(fe), \delta^{v_1, \dots, v_n}(fe), r_{u_1, \dots, u_n}^{v_1, \dots, v_n}(fe), fe_1 \bowtie fe_2 \in FE^\Sigma$;
6. if $\Phi \in Fr^\Sigma$ and $fe \in FE^\Sigma$, then $\sigma(\Phi, fe) \in FE^\Sigma$;
7. if $n \geq 1, \Phi \in Fr^\Sigma, fe_1, \dots, fe_n \in FE^\Sigma$, and $\bar{x} \in \bar{U}_n(V)$, then $S_P^{\bar{x}}(\Phi, fe_1, \dots, fe_n) \in Fr^\Sigma$;
8. if $n \geq 1, fe, fe_1, \dots, fe_n \in FE^\Sigma$, and $\bar{x} \in \bar{U}_n(V)$, then $S_F^{\bar{x}}(fe, fe_1, \dots, fe_n) \in FE^\Sigma$;
9. if $n \geq 1, prg \in Pt^\Sigma, fe_1, \dots, fe_n \in FE^\Sigma$, and $\bar{x} \in \bar{U}_n(V)$, then $S_F^{\bar{x}}(prg, fe_1, \dots, fe_n) \in Pt^\Sigma$;
10. if $x \in V^+$ and $fe \in FE^\Sigma$, then $AS^x(fe) \in Pt^\Sigma$;
11. $id \in Pt^\Sigma$;
12. if $prg_1, prg_2 \in Pt^\Sigma$, then $pr_1 \bullet pr_2 \in Pt^\Sigma$;
13. if $\Phi \in Fr^\Sigma$ and $prg_1, prg_2 \in Pt^\Sigma$, then $IF(\Phi, prg_1, prg_2) \in Pt^\Sigma$;
14. if $\Phi \in Fr^\Sigma$ and $prg \in Pt^\Sigma$, then $WH(\Phi, prg) \in Pt^\Sigma$.

The set $FHFr^\Sigma$ is the set of all formulas of the form $\{p\}f\{q\}$, where $p, q \in Fr^\Sigma$ and $f \in Pt^\Sigma$.

Definition 28. Let $\Sigma = (V, Ps, FEs, Prgs)$ be a logic signature and A be an arbitrary set. Then an interpretation J is a tuple $(RNPA(V, A), I_{Ps}, I_{FEs}, I_{Prgs})$, where $I_{Ps} : Ps \xrightarrow{t} Pr^{V,A}$ is an interpretation mapping for predicate symbols, $I_{FEs} : FEs \xrightarrow{t} FPr^{V,A}$ and $I_{Prs} : Prs \xrightarrow{t} FPr^{V,A}$ are interpretation mappings for function and program symbols, respectively.

For any interpretation $J = (RNPA(V, A), I_{Ps}, I_{FEs}, I_{Prgs})$ we will denote by J_{Fr} , J_{FE} , and J_{Pt} the formula, function, and program text interpretation mappings

$$\begin{aligned} J_{Fr} &: Fr^\Sigma \xrightarrow{t} Pr^{V,A}, \\ J_{FE} &: FE^\Sigma \xrightarrow{t} FPr^{V,A}, \\ J_{Pt} &: Pt^\Sigma \xrightarrow{t} FPr^{V,A} \end{aligned}$$

which are the standard extensions of I_{Ps} , I_{FEs} , and I_{Prgs} to Fr^Σ , FE^Σ , and Pt^Σ respectively (defined by structural induction). Also, we will denote by J_{FHF} the interpretation mapping of Floyd-Hoare assertions $J_{FHF} : FHF^\Sigma \xrightarrow{t} Pr^{V,A}$ defined as follows:

$$J_{FHF}(\{p\}f\{q\}) = FH(J_{Fr}(p), J_{Pt}(f), J_{Fr}(q)).$$

In this paper we will not define interpretations explicitly expecting that they are clear from the context. For any $P \in Fr^\Sigma$ or $P \in FHF^\Sigma$ we will denote by P_J or $(P)_J$ the predicate that corresponds to P under interpretation J . We will omit the index J when it is clear from the context.

We will use the following notation for any predicate p :

$$\begin{aligned} p^T &= \{d \mid p(d) \downarrow = T\} \text{ is the truth domain of a predicate } p; \\ p^F &= \{d \mid p(d) \downarrow = F\} \text{ is the falsity domain of } p. \end{aligned}$$

Definition 29. A formula $P \in Fr^\Sigma$ or a Floyd-Hoare assertion $P \in FHF^\Sigma$ is valid (irrefutable) in an interpretation J (denoted as $J \models P$), if $P_J^F = \emptyset$.

Definition 30. A formula $P \in Fr^\Sigma$ or a Floyd-Hoare assertion $P \in FHF^\Sigma$ is logically valid (denoted as $\models P$), if it is valid in every interpretation.

Let us define the logical consequence relation $\models \subseteq Fr^\Sigma \times Fr^\Sigma$ as

$$p \models q \Leftrightarrow \models p \rightarrow q,$$

where $p \rightarrow q$ means $\neg p \vee q$ for any $p, q \in Fr^\Sigma$.

We will also need the following special logical consequence relations

$$\models_T, \models_F \subseteq Fr^\Sigma \times Fr^\Sigma$$

such that

- $p \models_T q \Leftrightarrow p_J^T \subseteq q_J^T$ for every interpretation J ;
- $p \models_F q \Leftrightarrow q_J^F \subseteq p_J^F$ for every interpretation J .

5 Inference System for a Floyd-Hoare Logic with Partial Predicates

To make the program logic which we have defined applicable to software verification problems it is necessary to present an inference system. Such an inference system could be based on the inference system for the classical Floyd-Hoare logic with total predicates for the language WHILE [37], but it is known to be unsound in the case of partial predicates [8] which is considered in the paper. For this reason additional constraints need to be added to achieve a sound inference system.

We will write $\vdash_X p$ to denote that a formula p is *derived* in some inference system X . An inference system X is *sound*, if $\vdash_X p \Rightarrow \models p$ for each formula p , and is *complete*, if $\models p \Rightarrow \vdash_X p$ for each p . Completeness can be treated in extensional or intensional approaches. For *extensional completeness* [37] pre- and postconditions can be arbitrary predicates. *Intensional completeness* requires that pre- and postconditions are presented by formulas in a given language.

The classical inference system for the language WHILE [37] can be presented in semantic form as follows ($x \in V$):

$$\begin{array}{ll}
 R_AS \frac{}{\{S_P^x(p, h)\} AS^x(h) \{p\}} & R_SKIP \frac{}{\{p\} id \{p\}} \\
 R_SEQ \frac{\{p\} f \{q\}, \{q\} g \{r\}}{\{p\} f \bullet g \{r\}} & R_IF \frac{\{r \wedge p\} f \{q\}, \{\neg r \wedge p\} g \{q\}}{\{p\} IF(r, f, g) \{q\}} \\
 R_WH \frac{\{r \wedge p\} f \{p\}}{\{p\} WH(r, f) \{\neg r \wedge p\}} & R_CONS \frac{\{p'\} f \{q'\}}{\{p\} f \{q\}} p \rightarrow p', q' \rightarrow q
 \end{array}$$

This inference system is sound and extensionally complete for total predicates, but for partial predicates it is not sound [8, 9], because rules R_SEQ , R_WH , and R_CONS do not guarantee a valid derivation from valid premises.

There can be different solutions for this problem. Here we will restrict the class of assertions to the class of T -increasing assertions [9].

An assertion $\{p\}f\{q\}$ is T -increasing if $p \models_T PC(f, q)$. In this case the truth domain of p is included in the preimage of the truth domain of q under f .

It is important to note that for partial predicates T -increasing assertions are logically valid, i.e. $p \models_T PC(f, q)$ implies $\models \{p\}f\{q\}$.

Now we extend the inference system TI [9], oriented on T -increasing assertions, to the system RN . This extension takes into account that the language works with complex names and new superposition compositions appeared in our program language.

In this new system RN rules R_AS , R_SKIP , R_SEQ , R_IF , R_WH remain the same, but $v \in V^+$ in the rule R_AS since we consider complex names. To these rules the following new rules specifying superpositions into a program (procedure calls) are added:

$$\begin{array}{l}
 R_SFID \frac{}{\{S_P^x(p, g_1, \dots, g_n)\} S_F^x(id, g_1, \dots, g_n) \{p\}} \\
 R_SF \frac{\{p\} S_F^x(id, g_1, \dots, g_n) \bullet f \{q\}}{\{p\} S_F^x(f, g_1, \dots, g_n) \{q\}}
 \end{array}$$

Also, we have to change the consequence rule R_CONS to the following rule:

$$R_CONS' \frac{\{p'\} f \{q'\}}{\{p\} f \{q\}}, p \models_T p', q' \models_T q$$

Proposition 1. *The inference system RN is sound for the class of T -increasing assertions.*

To prove the proposition we should demonstrate that axioms specify T -increasing assertions, and that given T -increasing assertions as premises the rules specify T -increasing assertions as consequences.

These properties can be proved analogously to [8, Theorem 4].

Proposition 2. *The inference system RN is extensionally complete for the class of T -increasing assertions.*

The prove is analogous to the standard proofs based on the notion of weakest precondition (see, for example, [37]) but taking into account partiality of predicates we use preimage predicate transformer (preimage composition) instead of the weakest precondition. Such a proof is given in [9, Theorem 4.1]. To prove our proposition we should additionally consider rules for superpositions with complex names. Details are omitted here.

In the system RN a new unconventional consequence relation \models_T is used. Its main semantic properties were studied in [40]. Further investigation will permit to substitute this consequence relation by the corresponding inference relation \vdash_T . Let us also note that relational operations were not directly used in RN . The functions over relational data can be used inside arguments of assignment and superposition compositions. They will be specified explicitly in the predicate logic for relational nominative data. A detailed investigation of such logic is planned for the forthcoming publications.

6 Towards Formalization of Extended Floyd-Hoare Logic in Mizar

We proposed a formalization of nominative data in Mizar in [41–44]. In the mentioned work different types of nominative data were defined as Mizar *modes* with set parameters V and A which meant the sets of basic names and atomic values, respectively. We can use this formalization as a basis for formalization of the extended Floyd-Hoare logic for programs over relational nominative data. The following steps have to be done to achieve this.

- (1) Define the mode of binominative functions over $Nd(V, A)$. This gives us a formalization of $FPr_g^{V,A}$ defined above.
- (2) Similarly define the modes of partial predicates over $RND(V, A)$.
- (3) Formalize Pt^Σ (program texts), Fr^Σ (formulas), FE^Σ (functional expressions), and $FHFr^\Sigma$ (Floyd-Hoare assertions) as Mizar modes.

- (4) Formalize the interpretation mapping in accordance with Definition 28.
- (5) Formalize the relation of validity in an interpretation in accordance with Definition 29 and the relation of validity in accordance with Definition 30.
- (6) Formalize the logical consequence relation $p \models q$ and the special logical consequence relations $p \models_T q$ and $p \models_F q$ for $p, q \in Fr^\Sigma$.
- (7) Formulate the inference rules of the *RN* inference system described in Sect. 5 as Mizar schemes and formally prove their semantic validity.

Finally, the proven inference rules can be used to prove semantics properties of programs defined by concrete program texts (elements of Pt^Σ).

7 Related Work

Logical approaches to program specification and reasoning about program properties were used in the works by Floyd [1] and Hoare [2]. These approaches were based on axiomatic systems with total predicates and used triples of a precondition, program, and a postcondition. Later, it became evident that partiality of predicates and programs needs to be taken into account which gave rise to three-valued logics which represented undefinedness of a predicate by a special third value. In particular, such logics were studied by Lukasiewicz, Kleene, Bochvar and others. At the same time many other extensions of Floyd-Hoare logic were proposed as a basis for program verification, including Dynamic logic and Separation logic. In Dynamic logic [45] special modalities that allow usage of program texts and specifications alongside are used. A Floyd-Hoare assertion $\{p\}pr\{q\}$ can be replaced with a formula $p \rightarrow [pr]q$, where $[pr]q$ indicates that if a program pr terminates, then q necessarily holds. For deterministic programs, $[pr]q$ is equal to our preimage composition which also allows use of specifications and program texts together. Separation logic [46] was introduced to deal with widespread usage of heap and pointers in programming. This logic has special means for specifying heap properties. But only a heap function that maps memory addresses to values is assumed to be partial. In other aspects only total predicates are considered.

The ways of dealing with partiality in current software specification languages (VDM, RSL, Z, etc.) are described in [4]. Most approaches use either a many-valued logic or underspecification for dealing with partiality.

8 Conclusions

We have proposed an extension of Floyd-Hoare logic for the case of partial conditions and programs on hierarchically organized data called relational nominative data. Such data can conveniently represent many data structures used in programming. We have proposed a special inference system for our logic, investigated its soundness and extensional completeness and proposed an approach to its formalization in the Mizar system. In the future works we plan to implement the proposed approach and apply the results to software verification tasks.

References

1. Floyd, R.: Assigning meanings to programs. *Math. Asp. Comput. Sci.* **19**, 19–32 (1967)
2. Hoare, C.: An axiomatic basis for computer programming. *Commun. ACM* **12**(10), 576–580 (1969)
3. Apt, K.: Ten years of Hoare’s logic: a survey - part I. *ACM Trans. Program. Lang. Syst.* **3**(4), 431–483 (1981). <http://doi.acm.org/10.1145/357146.357150>
4. Hähnle, R.: Many-valued logic, partiality, and abstraction in formal specification languages. *Log. J. IGPL* **13**(4), 415–433 (2005). <http://dx.doi.org/10.1093/jigpal/jzi032>
5. Jones, C.: Reasoning about partial functions in the formal development of programs. In: *AVoCS 2005. Electronic Notes in Theoretical Computer Science*, vol. 145, pp. 3–25. Elsevier (2006). <https://doi.org/10.1016/j.entcs.2005.10.002>
6. Gries, D., Schneider, F.B.: Avoiding the undefined by underspecification. In: van Leeuwen, J. (ed.) *Computer Science Today. LNCS*, vol. 1000, pp. 366–373. Springer, Heidelberg (1995). <https://doi.org/10.1007/BFb0015254>
7. Duzi, M.: Do we have to deal with partiality? *Misc. Log.* **5**, 45–76 (2003)
8. 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, Cham (2013). https://doi.org/10.1007/978-3-319-03998-5_18
9. Nikitchenko, M., Kryvolap, A.: Properties of inference systems for Floyd-Hoare logic with partial predicates. *Acta Electrotech. Inform.* **13**(4), 70–78 (2013)
10. Skobelev, V., Nikitchenko, M., Ivanov, I.: On algebraic properties of nominative data and functions. In: Ermolayev, V., Mayr, H., Nikitchenko, M., Spivakovsky, A., Zholtkevych, G. (eds.) *Communications in Computer and Information Science, ICTERI*, vol. 469, pp. 117–138. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-13206-8_6
11. Nikitchenko, M., Shkilniak, S.: *Mathematical logic and theory of algorithms*. Publishing house of Taras Shevchenko National University of Kyiv, Ukraine (2008). (in Ukrainian)
12. Nikitchenko, M.: Composition-nominative aspects of address programming. *Cybern. Syst. Anal.* **45**, 864 (2009). <https://doi.org/10.1007/s10559-009-9159-4>
13. Wiedijk, F. (ed.): *The Seventeen Provers of the World*. Foreword by Dana S. Scott. *LNAI*, vol. 3600. Springer, Heidelberg (2006). <https://doi.org/10.1007/11542384>
14. Gordon, M.: Mechanizing programming logics in higher order logic. In: Birtwistle, G., Subrahmanyam, P. (eds.) *Current Trends in Hardware Verification and Automated Theorem Proving*, pp. 387–439. Springer, New York (1989). https://doi.org/10.1007/978-1-4612-3658-0_10
15. Von Wright, J., Hekanaho, J., Luostarinen, P., Langbacka, T.: Mechanizing some advanced refinement concepts. *Form. Methods Syst. Des.* **3**(1), 49–81 (1993). <https://doi.org/10.1007/BF01383984>
16. Bancerek, G., Byliński, C., Grabowski, A., Kornilowicz, A., Matuszewski, R., Naumowicz, A., Pał, K., Urban, J.: Mizar: state-of-the-art and beyond. In: Kerber, M., Carette, J., Kaliszyk, C., Rabe, F., Sorge, V. (eds.) *CICM 2015. LNCS (LNAI)*, vol. 9150, pp. 261–279. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-20615-8_17
17. Grabowski, A., Kornilowicz, A., Naumowicz, A.: Four decades of Mizar. *J. Autom. Reason.* **55**(3), 191–198 (2015). <http://dx.doi.org/10.1007/s10817-015-9345-1>

18. Trybulec, A.: Tarski Grothendieck set theory. *Formaliz. Math.* **1**(1), 9–11 (1990)
19. Naumowicz, A.: Interfacing external CA systems for Gröbner bases computation in Mizar proof checking. *Int. J. Comput. Math.* **87**(1), 1–11 (2010). <http://dx.doi.org/10.1080/00207160701864459>
20. Naumowicz, A.: SAT-enhanced MIZAR proof checking. In: Watt, S.M., Davenport, J.H., Sexton, A.P., Sojka, P., Urban, J. (eds.) *CICM 2014. LNCS (LNAI)*, vol. 8543, pp. 449–452. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-08434-3_37
21. Naumowicz, A.: Automating Boolean set operations in Mizar proof checking with the aid of an external SAT solver. *J. Autom. Reason.* **55**(3), 285–294 (2015). <http://dx.doi.org/10.1007/s10817-015-9332-6>
22. Kornilowicz, A.: On rewriting rules in Mizar. *J. Autom. Reason.* **50**(2), 203–210 (2013). <http://dx.doi.org/10.1007/s10817-012-9261-6>
23. Kornilowicz, A.: Flexary connectives in Mizar. *Comput. Lang. Syst. Struct.* **44**, 238–250 (2015). <http://dx.doi.org/10.1016/j.cl.2015.07.002>
24. Kornilowicz, A.: Definitional expansions in Mizar. *J. Autom. Reason.* **55**(3), 257–268 (2015). <http://dx.doi.org/10.1007/s10817-015-9331-7>
25. Nelson, G., Oppen, D.C.: Fast decision procedures based on congruence closure. *J. ACM* **27**, 356–364 (1980). <http://doi.acm.org/10.1145/322186.322198>
26. Grabowski, A., Kornilowicz, A., Schwarzweller, C.: Equality in computer proof-assistants. In: Ganzha, M., Maciaszek, L.A., Paprzycki, M. (eds.) *Proceedings of the 2015 FedCSIS. Annals of Computer Science and Information Systems*, vol. 5, pp. 45–54. IEEE (2015). <https://doi.org/10.15439/2015F229>
27. Naumowicz, A., Byliński, C.: Improving MIZAR texts with *properties* and *requirements*. In: Asperti, A., Bancerek, G., Trybulec, A. (eds.) *MKM 2004. LNCS*, vol. 3119, pp. 290–301. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-27818-4_21
28. Kornilowicz, A.: Enhancement of MIZAR texts with transitivity property of predicates. In: Kohlhase, M., Johansson, M., Miller, B., de Moura, L., Tompa, F. (eds.) *CICM 2016. LNCS (LNAI)*, vol. 9791, pp. 157–162. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-42547-4_12
29. Naumowicz, A., Kornilowicz, A.: Introducing Euclidean relations to Mizar. [47], pp. 245–248. <https://doi.org/10.15439/2017F368>
30. Grabowski, A.: Mechanizing complemented lattices within Mizar type system. *J. Autom. Reason.* **55**(3), 211–221 (2015). <http://dx.doi.org/10.1007/s10817-015-9333-5>
31. Grabowski, A., Kornilowicz, A., Schwarzweller, C.: On algebraic hierarchies in mathematical repository of Mizar. In: Ganzha, M., Maciaszek, L.A., Paprzycki, M. (eds.) *Proceedings of the 2016 FedCSIS. Annals of Computer Science and Information Systems*, vol. 8, pp. 363–371. IEEE (2016). <https://doi.org/10.15439/2016F520>
32. Grabowski, A., Jastrzębska, M.: Rough set theory from a math-assistant perspective. In: Kryszkiewicz, M., Peters, J.F., Rybinski, H., Skowron, A. (eds.) *RSEISP 2007. LNCS (LNAI)*, vol. 4585, pp. 152–161. Springer, Heidelberg (2007). https://doi.org/10.1007/978-3-540-73451-2_17
33. Grabowski, A., Schwarzweller, C.: On duplication in mathematical repositories. In: Autexier, S., Calmet, J., Delahaye, D., Ion, P.D.F., Rideau, L., Rioboo, R., Sexton, A.P. (eds.) *CICM 2010. LNCS (LNAI)*, vol. 6167, pp. 300–314. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-14128-7_26

34. Pak, K.: Improving legibility of natural deduction proofs is not trivial. *Log. Methods Comput. Sci.* **10**(3), 1–30 (2014). [http://dx.doi.org/10.2168/LMCS-10\(3:23\)2014](http://dx.doi.org/10.2168/LMCS-10(3:23)2014)
35. Pak, K.: Automated improving of proof legibility in the Mizar system. [48], pp. 373–387. https://doi.org/10.1007/978-3-319-08434-3_27
36. Pak, K.: Improving legibility of formal proofs based on the close reference principle is NP-hard. *J. Autom. Reason.* **55**(3), 295–306 (2015). <http://dx.doi.org/10.1007/s10817-015-9337-1>
37. Nielson, H., Nielson, F.: *Semantics with Applications - A Formal Introduction*. Wiley, Hoboken (1992)
38. Nikitchenko, M., Tymofieiev, V.: Satisfiability in composition-nominative logics. *Cent. Eur. J. Comput. Sci.* **2**(3), 194–213 (2012). <https://doi.org/10.2478/s13537-012-0027-3>
39. Dijkstra, E.: *A Discipline of Programming*, 1st edn. Prentice Hall PTR, Upper Saddle River (1997)
40. Nikitchenko, M., Shkilniak, S.: Semantic properties of T-consequence relation in logics of quasiary predicates. *Comput. Sci. J. Mold.* **23**(2(68)), 102–122 (2015)
41. Ivanov, I., Kornilowicz, A., Nikitchenko, M.: Formalization of nominative data in Mizar. In: *Proceedings of TAAPSD 2015*, pp. 82–85. Taras Shevchenko National University of Kyiv, Ukraine, 23–26 December 2015
42. Ivanov, I., Nikitchenko, M., Kryvolap, A., Kornilowicz, A.: Simple-named complex-valued nominative data - definition and basic operations. *Formaliz. Math.* **25**(3), 205–216 (2017). <http://dx.doi.org/10.1515/forma-2017-0020>
43. Kornilowicz, A., Kryvolap, A., Nikitchenko, M., Ivanov, I.: Formalization of the algebra of nominative data in Mizar. [47], pp. 237–244. <https://doi.org/10.15439/2017F301>
44. Kornilowicz, A., Kryvolap, A., Nikitchenko, M., Ivanov, I.: Formalization of the nominative algorithmic algebra in Mizar. In: Świątek, J., Borzowski, L., Wilimowska, Z. (eds.) *ISAT 2017. AISC*, vol. 656, pp. 176–186. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-67229-8_16
45. Harel, D., Tiuryn, J., Kozen, D.: *Dynamic Logic*. MIT Press, Cambridge (2000)
46. Reynolds, J.C.: Separation logic: a logic for shared mutable data structures. In: *Proceedings of the 17th LICS*, pp. 55–74 (2002)
47. Ganzha, M., Maciaszek, L.A., Paprzycki, M. (eds.): *Proceedings of FedCSIS 2017. Annals of Computer Science and Information Systems*, vol. 11. IEEE, Prague, Czech Republic, 3–6 September 2017
48. Watt, S.M., Davenport, J.H., Sexton, A.P., Sojka, P., Urban, J. (eds.): *CICM 2014. LNCS (LNAI)*, vol. 8543. Springer, Cham (2014). <https://doi.org/10.1007/978-3-319-08434-3>