An Approach To Formalization of an Extension of Floyd-Hoare Logic

Artur Korniłowicz[1], Andrii Kryvolap[2], Mykola Nikitchenko[2], Ievgen Ivanov[2]

[1]Institute of Informatics, University of Białystok,
Ciołkowskiego 1M, 15-245 Białystok, Poland
[2]Taras Shevchenko National University of Kyiv
64/13, Volodymyrska Street, 01601 Kyiv, Ukraine
arturk@math.uwb.edu.pl, krivolapa@gmail.com,
nikitchenko@unicyb.kiev.ua, ivanov.eugen@gmail.com

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 on hierarchically organized data. These data are based on the naming relation and are called 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 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.

Key Terms. Specification Process, Verification Process

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 hierarchically organized data. As was shown in [10] hierarchically organized data (more specifically, a class of such data called nominative data [10]) is sufficient for representing many data structures (like multidimensional arrays, lists, trees, tables, etc.) that are frequently used in programming. 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 [11], 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* [2] (a special class of hierarchically organized data) are adequate mathematical models of various forms of data that are processed and stored in computing systems. There exist several types of nominative data [11] (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.

The relation of the above mentioned notions to semantics of programming languages can be illustrated on a simple programming language WHILE [12]. This is an imperative language in which programs are composed from statements which contain boolean and arithmetic expressions. A program state is an assignment of values to variable names which can be modeled as a nominative set. Semantics of a statement can be represented as a partial binominative function (a mapping from states to states) and semantics of a boolean or arithmetic expression can be represented as a function on nominative data which takes boolean or integer values. Semantics of statements are composed to obtain semantics of a program.

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 the carrier sets of our program algebra will consist of partial functions and predicates over nominative data with complex names and complex values [10].

We will treat a Floyd-Hoare triple as a composition with two predicates on 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, predicate 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 predicate 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) and has one of the richest libraries of formalized mathematical theories that spans many branches of mathematics. This library is called Mizar Mathematical Library (MML). 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 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 the 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 [12] is sound and extensionally complete for the classical Floyd-Hoare logic with total predicates [12] (extensional completeness means that pre- and postconditions may be arbitrary predicates [12]; 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 completes problem 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 a Floyd-Hoare logic with partial predicates over flat (nonhierarchical) data obtained in [8, 9].

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

2 Algebra of 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]. Before giving 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 graph. 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 $d \in D$;
- $\mathrm{dom}(f) = \{d \in D \mid f(d)\downarrow\}$ is the domain of a function (note that in different branches of mathematics there exist different definitions of the domain of a partial functions; 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 A the corresponding class ${}^{V}A$ of *nominative sets* is defined as

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

We will use the following notations for nominative sets:

- $[v_1 \to a_1, \ldots, v_n \to a_n]$, where v_1, ..., v_n are names from V and a_1, ... a_n are atoms from A, denotes a nominative set with the graph $\{(v_1, a_1), \ldots, (v_n, a_n)\}$;
- $[v_i \mapsto a_i | i \in I]$, 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;
- $[]$ or $\emptyset$ denotes the empty nominative set (a nowhere defined function).

Nominative data are built over classes of names V and basic values (atoms) A using naming relations *name→value*. A rough idea is that a nominative data d is either an atom from A, or has the form $[v_1 \to d_1, \ldots, v_n \to d_n]$, where v_1, ..., v_n are names from V and d_1, ..., d_n are either atoms or other nominative data.

Nominative data are classified in accordance with the following 2 parameters [10]: *values* can be simple (unstructured) or complex (structured), and *names* can be simple (unstructured) or complex (structured). These parameters give 4 types of nominative data. 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.

The definitions of 4 types of nominative data (TND_{SS}, TND_{SC}, TND_{CS}, TND_{CC}) are given below. 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*.

1. Data of the type TND_{SS} (*data with simple names and simple values*) are elements of the set $V \xrightarrow{n} A$. For example,

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

   if $u, v \in V$, $1, 2 \in A$.

2. Data of the type TND_{SC} (*data with simple names and complex values*) are elements of the set $ND(V, A)$, where

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

$$ND_0(V, A) = A \cup \{\emptyset\},\ ND_{k+1}(V, A) = ND_k(V, A) \cup \left(V \xrightarrow{n} ND_k(V, A)\right)$$

for all $k = 0, 1, 2, ...$ Data of this class are hierarchically constructed, for example,

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

if $u, v, w \in V$ and $1, 2 \in A$.
Such data can be represented by oriented trees with arcs labeled by names and leafs labelled by atoms. We will call any finite sequence of names $p = (v_1, v_2, ..., v_k)$ a *path*. A *path in a given data* $d \in ND(V, A)$ is a path $(v_1, v_2, ..., v_k)$ such that the value of the expression $(...((d(v_1))(v_2))...(v_k))$ is defined (it corresponds to a path from the root to a leaf in a tree). If $p = (v_1, v_2, ..., v_k)$ is a path in d, we will say that $(...((d(v_1))(v_2))...(v_k))$ is the value of p in d and denote it as $d(v_1, v_2, ..., v_k)$. A *terminal path* is a path with atomic or empty value. The *rank* of d is the least k such that $d \in ND_k(V, A)$.
3. Data of the type TND_{CS} (*data with complex names and simple values*) are elements of the set $NDVS(V, A)$, where $NDVS(V, A)$ is the set of all elements of

$$A \cup (V^+ \xrightarrow{n} A)$$

such that either $d \in A$, or $d \in V^+ \xrightarrow{n} A$ and all strings from $\mathrm{dom}(d)$ are pairwise incomparable in the sense of the prefix relation. For example,

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

if $u, v, w \in V$ are different and $1, 2, 3 \in A$.
4. Data of the type TND_{CC} (*data with complex names and complex values*) are elements of the set $NDVC(V, A)$, where $NDVC(V, A)$ is the set of all $d \in ND(V^+, A)$ such that for any two paths $(u_1, u_2, ..., u_k)$ and $(v_1, v_2, ..., v_l)$ in d, neither of which is a prefix of another, words $u_1u_2...u_k$ and $v_1v_2...v_l$ are incomparable in the sense of prefix relation (*principle of unambiguous associative naming*). For example,

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

if $u, v, w \in V$ are different, $1, 2 \in A$.

In [8] it was shown how conventional data structures can be represented by different kinds of nominative data. The most complex and interesting type of nominative data is TND_{CC}, so we will use it in this paper.

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

Below we give definitions of these operations for nominative data of the type TND_{CC}. For any word $u \in V^+$ and any data $d \in NDVC(V, A)$ 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* $v \Rightarrow_a : NDVC(V, A) \xrightarrow{p} NDVC(V, A)$ *is an operation with a parameter* $v \in V^+$ *defined by induction on 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* $d/v \neq \emptyset$, *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)

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

for all complex names $u, u_1, u_2, ..., u_n \in V^+$ such that $u = u_1u_2...u_n$.

Definition 2. *Naming is an operation* $\Rightarrow v : NDVC(V, A) \xrightarrow{t} NDVC(V, A)$ *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 procedures calls and the local overlapping formalizes the assignment operator in programming languages.

Definition 3. *Global overlapping is a binary operation*

$$\nabla_a : NDVC(V, A) \times NDVC(V, A) \xrightarrow{p} NDVC(V, A)$$

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

Let $NDVC_k(V, A) = NDVC(V, A) \cap ND_k(V, A)$ *(data of the rank not greater than k).*

Induction base. If $d_0 \in NDVC_0(V, A)$*, then*

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

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

- $d(u) = d_2(u)$*, if* $u \in \mathrm{dom}(d_2)$ *and* u *does not have a proper prefix which belongs to* $\mathrm{dom}(d_1)$*.*
- $d(u) = d_1(u) \nabla_a (d_2/u)$*, if* $d_1(u) \downarrow$ *and* $d_1(u) \notin A$*, and* u *is a proper prefix of some element of* $\mathrm{dom}(d_2)$*;*
- $d(u) = d_2/u$*, if* $d_1(u) \downarrow$ *and* $d_1(u) \in A$*, and* u *is a proper prefix of some element of* $\mathrm{dom}(d_2)$*;*
- $d(u) = d_1(u)$*, if* $d_1(u) \downarrow$ *and* u *is not comparable (in the sense of prefix relation) with any element of* $\mathrm{dom}(d_2)$*;*
- $d(u) \uparrow$*, 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 under a name in the second argument overwrites a value under extension of this name in the first argument;
3. $[u \mapsto d_1] \nabla_a [uv \mapsto d_2] = [u \mapsto (d_1 \nabla_s [v \mapsto d_2])]$ $(d_1 \notin A)$, i.e., a value under a name in the second argument modifies values under prefixes of this name in the first argument.

Definition 4. *Local overlapping is an operation* $\nabla_a^v : NDVC(V, A) \times NDVC(V, A) \xrightarrow{p} NDVC(V, A)$ *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,

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

Definition 5. *An algebra of nominative data of type* TND_{CC} *(denoted as* $NDA(V, A)$*) is an algebra with the carrier* $NDVC(V, A)$ *and the operations* $\{\Rightarrow v\}_{v \in V^+}$*,* $\{v \Rightarrow_a\}_{v \in V^+}$*,* $\{\nabla_a^v\}_{v \in V^+}$*.*

3 Semantics of Extended Floyd-Hoare Logic

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

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

Multi-sorted algebras on sets of partial nominative predicates and partial binominative functions can be used to define semantics of program logics [11, 18]. 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:

- 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.

We also need compositions that provide the possibility to construct predicates describing properties of programs. The main composition of this kind is 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 [19].

Let us give definitions of the mentioned compositions.

Definition 6. *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* $d \in NDVC(V, A)$*:*

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

Definition 7. *Negation is a unary composition*

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

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

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

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

Definition 8. *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 $d \in NDVC(V,A)$:

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

For each $n = 1, 2, 3, ...$ denote by $\bar{U}_n(V)$ the set of all tuples $(x_1, ..., x_n) \in (V^+)^n$ of n complex names such that $x_1, x_2, ..., x_n$ 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 9. *For each $n = 1, 2, 3, ...$, superposition of n functions into a function is a n+1-ary composition*

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

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

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

Definition 10. *For each $n = 1, 2, 3, ...$, superposition of n functions into a predicate is a n+1-ary composition*

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

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

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

Definition 11. *Denomination is a null-ary composition $'x : FPrg^{V,A}$ with a parameter $x \in V^+$ such that for each $d \in NDVC(V,A)$:*

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

Definition 12. *Assignment over hierarchical data is a composition*

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

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

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

Definition 13. *Identity program composition is a null-ary composition* $id : FPrg^{V,A}$ *such that for each* $d \in NDVC(V, A)$*:*

$$id(d) = d.$$

Definition 14. *Sequential execution is a binary composition*

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

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

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

Definition 15. *Branching is a ternary composition*

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

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

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

Definition 16. *While cycle is a binary composition*

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

such that for each $r \in Pr^{V,A}$ *(condition),* $f \in FPrg^{V,A}$ *(loop body), and* $d \in NDVC(V, A)$*:*

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

if there exists an integer $n \geq 0$ *such that* $p(f^{(i)}(d)) \downarrow = T$ *for all* $i = 0, 1, ..., n-1$ *and* $p(f^{(n)}(d)) \downarrow = F$*, where* $f^{(i)}$ *denotes* $\underbrace{f \bullet f \bullet ... \bullet f}_{i}$ *and* $f^{(0)} = id$*;*

and $WH(p, f)(d) \uparrow$*, otherwise.*

Definition 17. *Monotone Floyd-Hoare composition*

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

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

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

Definition 18. *Predicate transformer composition is a binary composition*

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

such that for all $q \in Pr^{V,A}$, $f \in FPrg^{V,A}$, *and* $d \in NDVC(V,A)$*:*

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

Predicate transformer composition is the same as *Glushkov prediction operation* (sequential execution of a function and a predicate) [10]. We call this composition (defined for partial predicates) as preimage predicate transformer composition in order to relate it to the *weakest precondition* predicate transformer [19]. Note that $FH(p,pr,q) = p \to PC(pr,q)$.

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.

Definition 19. *An (associative) nominative program algebra* $NPA(V,A)$ *is a two-sorted algebra*

$$< Pr^{V,A}, FPrg^{V,A};\ \vee, \neg, \{\exists x\}_{x\in V^+}, \{S_P^{\bar{x}}\}_{\bar{x}\in \bar{U}(V)}, \{S_F^{\bar{x}}\}_{\bar{x}\in \bar{U}(V)}, \{'x\}_{x\in V^+}, id,$$

$$\{AS^x\}_{x\in V^+}, \bullet, IF, WH, FH, PC > .$$

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

Terms of the algebra $NPA(V,A)$ defined over some sets of predicate symbols Ps and program symbols Prs specify the syntax (language) of the logic. Let us give definitions of the sets of program texts $Pt^{V,Ps,Prs}$, formulas $Fr^{V,Ps,Prs}$, and Floyd-Hoare assertions $FHFr^{V,Ps,Prs}$.

The sets $Pt^{V,Ps,Prs}$ and $Fr^{V,Ps,Prs}$ are defined inductively (here we use the symbols of compositions in the purely syntactic sense, i.e. they are currently not associated with semantics):

1. if $pr \in Prs$, then $pr \in Pt^{V,Ps,Prs}$;
2. if $n \geq 1$, $pr, pr_1, ..., pr_n \in Prs$, and $\bar{x} \in \bar{U}_n(V)$, then $S_F^{\bar{x}}(pr, pr_1, ..., pr_n) \in Pt^{V,Ps,Prs}$;
3. if $x \in V^+$ and $pr \in Prs$, then $AS^x(pr) \in Pt^{V,Ps,Prs}$;

4. $'x \in Pt^{V,Ps,Prs}$ for each $x \in V^+$ and $id \in Pt^{V,Ps,Prs}$;
5. if $pr_1, pr_2 \in Pt^{V,Ps,Prs}$, then $pr_1 \bullet pr_2 \in Pt^{V,Ps,Prs}$;
6. if $\Phi \in Fr^{V,Ps,Prs}$ and $pr_1, pr_2 \in Pt^{V,Ps,Prs}$, then $IF(\Phi, pr_1, pr_2) \in Pt^{V,Ps,Prs}$;
7. if $\Phi \in Fr^{V,Ps,Prs}$ and $pr \in Pt^{V,Ps,Prs}$, then $WH(\Phi, pr) \in Pt^{V,Ps,Prs}$;
8. if $P \in Ps$, then $P \in Fr^{V,Ps,Prs}$;
9. if $\Phi, \Psi \in Fr^{V,Ps,Prs}$, then $\Phi \vee \Psi, \Phi \wedge \Psi \in Fr^{V,Ps,Prs}$;
10. if $\Phi \in Fr^{V,Ps,Prs}$, then $\neg\Phi \in Fr^{V,Ps,Prs}$;
11. if $n \geq 1$, $\Phi \in Fr^{V,Ps,Prs}$, $pr_1, ..., pr_n \in Prs$, and $\bar{x} \in \bar{U}_n(V)$, then $S^{\bar{x}}_P(\Phi, pr_1, ..., pr_n) \in Fr^{V,Ps,Prs}$;
12. if $\Phi \in Fr^{V,Ps,Prs}$ and $v \in V^+$, then $\exists v\Phi \in Fr^{V,Ps,Prs}$.

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

Definition 20. *Let V be a set of basic names, Ps be a set of predicate symbols, Prs be a set of program symbols, and A be an arbitrary set. Then an interpretation J is a triple $(NDA(V, A), I_{Ps}, I_{Prs})$, where $I_{Ps} : Ps \xrightarrow{t} Pr^{V,A}$ is an interpretation mapping for predicate symbols and $I_{Prs} : Prs \xrightarrow{t} FPrg^{V,A}$ is an interpretation mapping for program symbols.*

For any interpretation $J = (NDA(V, A), I_{Ps}, I_{Prs})$ we will denote by J_{Fr} and J_{Pt} the *formula and program text interpretation mappings*

$$J_{Fr} : Fr^{V,Ps,Prs} \xrightarrow{t} Pr^{V,A},$$

$$J_{Pt} : Pt^{V,Ps,Prs} \xrightarrow{t} FPrg^{V,A}$$

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

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

In this paper we will not define interpretations explicitly expecting that they are clear from the context. For any $P \in Fr^{V,Ps,Prs}$ or $P \in FHFr^{V,Ps,Prs}$ 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:
$p^T = \{d \mid p(d) \downarrow = T\}$ is the truth domain of a predicate p;
$p^F = \{d \mid p(d) \downarrow = F\}$ is the falsity domain p.

Definition 21. *A formula $P \in Fr^{V,Ps,Prs}$ or a Floyd-Hoare assertion $P \in FHFr^{V,Ps,Prs}$ is valid (irrefutable) in an interpretation J (denoted as $J \models P$), if $P^F_J = \emptyset$.*

Definition 22. *A formula $P \in Fr^{V,Ps,Prs}$ or a Floyd-Hoare assertion $P \in FHFr^{V,Ps,Prs}$ is valid (denoted as $\models P$), if it is valid in every interpretation.*

Let us define the *logical consequence relation* $\models \subseteq Fr^{V,Ps,Prs} \times Fr^{V,Ps,Prs}$:

$$p \models q \Leftrightarrow \models p \to q,$$

where $p \to q$ means $\neg p \vee q$ for any $p, q \in Fr^{V,Ps,Prs}$.

We will also need the following *special logical consequence relations*

$$\models_T, \models_F \subseteq Fr^{V,Ps,Prs} \times Fr^{V,Ps,Prs}$$

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 over Nominative data

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 [12], but it is known be unsound in the case of partial predicates [9] 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* [9] 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 [12] can be presented in semantic form as follows ($x \in V$):

$$R\_AS\ \frac{}{\{S_P^x(p,h)\}\ AS^x(h)\ \{p\}} \qquad R\_SKIP\ \frac{}{\{p\}\ id\ \{p\}}$$

$$R\_SEQ\ \frac{\{p\}\ f\ \{q\}, \{q\}\ g\ \{r\}}{\{p\}\ f \bullet g\ \{r\}} \qquad R\_IF\ \frac{\{r \wedge p\}\ f\ \{q\}, \{\neg r \wedge p\}\ g\ \{q\}}{\{p\}\ IF(r,f,g)\ \{q\}}$$

$$R\_WH\ \frac{\{r \wedge p\}\ f\ \{p\}}{\{p\}\ WH(r,f)\ \{\neg r \wedge p\}} \qquad R\_CONS\ \frac{\{p'\}\ f\ \{q'\}}{\{p\}\ f\ \{q\}}\ p \to p', q' \to q$$

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

As one of solutions we propose the following inference system AC with added constraints ($x \in V^+$, $n \geq 1$, $\bar{x} \in \bar{U}_n$):

$$R\_AS'\ \frac{}{\{S_P^x(p,h)\}\ AS^x(h)\ \{p\}}$$

$$R\_SFID'\ \frac{}{\{S_P^{\bar{x}}(p,g_1,...,g_n)\} S_F^{\bar{x}}(id,g_1,...,g_n)\{p\}}$$

$$R\_SF' \ \frac{\{p\}S_F^{\bar{x}}(id, g_1, ..., g_n) \bullet f\{q\}}{\{p\}S_F^{\bar{x}}(f, g_1, ..., g_n)\{q\}}$$

$$R\_SKIP' \ \frac{}{\{p\}\ id\ \{p\}}$$

$$R\_SEQ' \frac{\{p\}\ f\ \{q\}, \{q\}\ g\ \{r\}}{\{p\}\ f \bullet g\ \{r\}}\ , p \models PC(f \bullet g, r)$$

$$R\_IF' \ \frac{\{r \wedge p\}\ f\ \{q\}, \{\neg r \wedge p\}\ g\ \{q\}}{\{p\}\ IF(r, f, g)\ \{q\}}$$

$$R\_WH' \frac{\{r \wedge p\}\ f\ \{p\}}{\{p\}\ WH(r, f)\ \{\neg r \wedge p\}}\ , p \models PC(WH(r, f), \neg r \wedge p)$$

$$R\_CONS' \frac{\{p'\}\ f\ \{q'\}}{\{p\}\ f\ \{q\}}\ , p \models_T p', q' \models_F q$$

In this system constraints are presented via consequence relations $\models$, $\models_T$, $\models_F$. These relations can be substituted by the corresponding inference relations ($\vdash$, $\vdash_T$, $\vdash_F$), but this question is outside the scope of the paper and we will not go into detail here.

Theorem 1. $\vdash_{AC} \{PC(pr, q)\}pr\{q\}$ *for each* pr *and* q.

Proof (presented for some arbitrary interpretation J*, but for simplicity sake we omit this* J *from the text of this proof).*

Let us use induction over the structure of pr.

Base of induction. Let us prove that $\vdash_{AC} \{PC(AS^x(h), q)\}AS^x(h)\{q\}$. By the rule *R_AS'*, it is sufficient to show that $PC(AS^x(h), q) = S_P^x(q, h)$. Using the definition of PC and assignment it is easy to check that $PC(AS^x(h), q)(d) = q(d\nabla_a^x h(d))$. Moreover, we have: $S_P^x(q, h) = q(d\nabla_a[x \mapsto h(d)]) = q(d\nabla_a^x h(d)) = PC(AS^x(h), q)$. Thus

$$\vdash_{AC} \{PC(AS^x(h), q)\}AS^x(h)\{q\}.$$

For $\vdash_{AC} \{PC(id, q)\}id\{q\}$ the proof is obvious.

Inductive step.

1. The case when pr is $S_F^{\bar{x}}(f, g_1, ..., g_n)$ is straightforward.
2. Consider the case of sequential execution. Given the premises

   $$\vdash_{AC} \{PC(pr_1, PC(pr_2, q))\}pr_1\{PC(pr_2, q)\}\ , \vdash_{AC} \{PC(pr_2, q)\}pr_2\{q\}\ ,$$

   the required derivation $\vdash_{AC} \{PC(pr_1 \bullet pr_2, q)\}pr_1 \bullet pr_2\{q\}$ can be proved in the same way as in the proof of the sequential execution case in [9, Theorem 4.2], so we omit a detailed proof here.
3. Consider the branching composition. Assume that

   $$\vdash_{AC} \{PC(pr_1, q)\}pr_1\{q\},\ \vdash_{AC} \{PC(pr_2, q)\}pr_2\{q\}.$$

Let us prove that $\vdash_{AC} \{PC(IF(r,pr_1,pr_2),q)\}IF(r,pr_1\ ,pr_2)\{q\}$.
Let us prove $\neg r \wedge PC(IF(r,pr_1,pr_2),q) \models_T PC(pr_2,q)$.
If $d \in (\neg r \wedge PC(IF(r,pr_1,pr_2),q))^T$, then we have $r(d)\downarrow = F$, $IF(r,pr_1,pr_2)(d)\downarrow$, and $q(IF(r,pr_1,pr_2)(d)) \downarrow = T$. Then $IF(r,pr_1,pr_2)(d) \downarrow = pr_2(d)$. Thus $pr_2(d)\downarrow$ holds and $q(pr_2(d))\downarrow = T$, so $PC(pr_2,q)(d)\downarrow = T$, so $d \in PC(pr_2,q)^T$. Thus

$$\neg r \wedge PC(IF(r,pr_1,pr_2),q) \models_T PC(pr_2,q).$$

Similarly, $r \wedge PC(IF(r,pr_1,pr_2),q) \models_T PC(pr_1,q)$. Then by *R_CONS'*

$$\vdash_{AC} \{r \wedge PC(IF(r,pr_1,pr_2),q)\}pr_1\{q\}\ ,$$

$$\vdash_{AC} \{\neg r \wedge PC(IF(r,pr_1,pr_2),q)\}pr_2\{q\}\ .$$

Then using *R_IF* we conclude that

$$\vdash_{AC} \{PC(IF(r,pr_1,pr_2),q)\}IF(r,pr_1,pr_2)\{q\}\ .$$

4. Let us prove $\vdash_{AC} \{PC(WH(r,pr),q)\}WH(r,pr)\{q\}$ for each pr,q,r.
Let $p = PC(WH(r,pr),q)$. Let p' be a predicate such that: $p'(d) = T$, if $p(d)\downarrow = T$ and $p'(d) = F$ otherwise. We have $\vdash_{AC} \{PC(pr,p')\}pr\{p'\}$.
Let us show that $r \wedge p' \models_T PC(pr,p')$. Note that $p^T = p'^T$, whence $p \models_T p'$.
Assume that $(r \wedge p')(d)\downarrow = T$. Then $p'(d)\downarrow = p(d) = T$, whence $q(WH(r,pr)(d))\downarrow = T$. Moreover, $r(d)\downarrow = T$, so there exists $d' = pr(d)$ such that $WH(r,pr)(d')\downarrow = WH(r,pr)(d)$ and $q(WH(r,pr)(d'))\downarrow = T$. Then

$$p'(d')\downarrow = p(d') = PC(WH(r,pr),q)(d') = T.$$

From $p'(d')\downarrow = T$ and $d' = pr(d)$ we have $PC(pr,p')(d)\downarrow = T$. We conclude that $r \wedge p' \models_T PC(pr,p')$.
Let us prove that $\neg r \wedge p' \models_F q$. Assume that $q(d)\downarrow = F$.
If $r(d)\downarrow = T$ then $(\neg r \wedge p')(d)\downarrow = F$.
If $r(d)\downarrow = F$, then $p(d) = PC(WH(r,pr),q)(d)\downarrow = F$, so $p'(d) = p(d)\downarrow = F$ and $(\neg r \wedge p')(d)\downarrow = F$.
If $r(d)\uparrow$, then $p(d) = PC(WH(r,pr),q)(d)\uparrow$, whence $p'(d)\downarrow = F$ and $(\neg r \wedge p')(d)\downarrow = F$.
In all cases, $(\neg r \wedge p')(d)\downarrow = F$. Thus $\neg r \wedge p' \models_F q$.
Let us show that $p' \models_T PC(WH(r,pr),\neg r \wedge p')$. Assume that $p'(d)\downarrow = T$. Then $p(d)\downarrow = T$. Then there exists d' such that $WH(r,pr)(d)\downarrow = d'$ and $q(d') = T$, because $p = PC(WH(r,pr),q)$. By the definition of WH, $r(d')\downarrow = F$ and $WH(r,pr)(d')\downarrow = d'$. Because $WH(r,pr)(d')\downarrow = d'$ and $q(d') = T$, we have $p(d')\downarrow = T = PC(WH(r,pr),q)(d')$. Then $p'(d')\downarrow = T$ and $(\neg r \wedge p')(d')\downarrow = T$. Then $WH(r,pr)(d)\downarrow = d'$ and $PC(WH(r,pr),\neg r \wedge p')(d)\downarrow = T$. We conclude that $p' \models_T PC(WH(r,pr),\neg r \wedge p')$.
Then $p' \models PC(WH(r,pr),\neg r \wedge p')$. Moreover, from $r \wedge p' \models_T PC(pr,p')$ and *R_CONS'*, $\vdash_{AC} \{r \wedge p'\}pr\{p'\}$, so $\vdash_{AC} \{p'\}WH(r,pr)\{\neg r \wedge p'\}$ by the rule *R_WH'*. Because $\neg r \wedge p' \models_F q$ and $p \models_T p'$, we have $\vdash_{AC} \{p\}WH(r,pr)\{q\}$ by *R_CONS'*. Thus

$$\vdash_{AC} \{PC(WH(r,pr),q)\}WH(r,pr)\{q\}\ .$$

We have considered all compositions, so, taking into account that an interpretation J was chosen arbitrary, we have that $\vdash_{AC} \{PC(pr, q)\}pr\{q\}$ for each pr and q.

Theorem 2. *The inference system AC is sound.*

This theorem can be proved analogously to [8, Theorem 4]. The completeness problems (extensional and intensional) are more difficult. In this case it is reasonable to identify different classes of predicates (similarly to [8, 9]) and to prove completeness for such classes of predicates. In this case Theorem 1 is used.

6 Towards Formalization of Extended Floyd-Hoare Logic in Mizar

We proposed a formalization of nominative data in Mizar in [20]. 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 nominative data of the types TND_{CC} and TND_{SC}.

1) Define the mode of binominative functions over nominative data of the type TND_{SC} (please refer to [20] for the definition of `TypeSCNominativeData` mentioned below):

```
reserve V for set;
reserve A for set;
reserve D for TypeSCNominativeData of V, A;
definition
  let V, A, D;
  mode TypeSCBinominativeFunction of D is PartFunc of D, D;
end;
```

and, similarly, the mode of binominative functions over nominative data of the type TND_{CC}. This gives us a formalization of $FPrg^{V,A}$ defined in Section 3.

2) Analogously define the modes of partial predicates over nominative data of the types TND_{SC}, TND_{CC} over V, A ($Pr^{V,A}$).

3) Formalize $Pt^{V,Ps,Prs}$ (program texts), $Fr^{V,Ps,Prs}$ (formulas), $FHFr^{V,Ps,Prs}$ (and Floyd-Hoare assertions) as Mizar modes with parameters V, Ps, Prs.

4) Formalize the interpretation mapping in accordance with Definition 20.

5) Formalize the relation of validity in an interpretation in accordance with Definition 21 and the relation of validity in accordance with Definition 22.

6) Formalize the logical consequence relation $p \models q \Leftrightarrow \models p \to q$ for $p, q \in Fr^{V,Ps,Prs}$ and the special logical consequence relations $p \models_T q \Leftrightarrow p_J^T \subseteq q_J^T$ for any interpretation J, and $p \models_F q \Leftrightarrow q_J^F \subseteq p_J^F$ for any interpretation J.

7) Formulate the inference rules of the AC inference system described in Section 5 as Mizar schemes and 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 $Pt^{V,Ps,Prs}$).

7 Related Work

Logical approaches to program specification and reasoning about program properties were used in the works by R. Floyd [1] and T. 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 the Floyd-Hoare logic were proposed as a basis for program verification, including Dynamic logic and Separation logic. In Dynamic logic [21] 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 $p \to [pr]q$, where $[pr]q$ indicates that if a program pr terminates, then necessarily holds q. For deterministic programs, $[pr]q$ is equal to our preimage composition which also allows use of specifications and program texts together. Separation logic [22] 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 the Floyd-Hoare logic for the case of partial conditions and programs on hierarchically organized data called nominative data. Such data can be used to conveniently represent many data structures used in programming. We have proposed an inference system for our extended Floyd-Hoare logic and investigated its soundness and extensional completeness and propose 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. Mathematical aspects of computer science **19** (1967)
2. Hoare, C.: An axiomatic basis for computer programming. Commun. ACM **12** (1969) 576–580
3. Apt, K.: Ten years of Hoare's logic: A survey – part I. ACM Trans. Program. Lang. Syst. **3** (1981) 431–483
4. Hähnle, R.: Many-valued logic, partiality, and abstraction in formal specification languages. Logic Journal of the IGPL **13** (2005) 415–433

5. Jones, C.: Reasoning about partial functions in the formal development of programs. In: Proceedings of AVoCS'05. Volume 145., Elsevier, Electronic Notes in Theoretical Computer Science (2006) 3–25
6. Gries, D., Schneider, F.: Avoiding the undefined by underspecification. Technical report, Ithaca, NY, USA (1995)
7. Duzi, M.: Do we have to deal with partiality? Miscellanea Logica **5** (2003) 45–76
8. Kryvolap, A., Nikitchenko, M., Schreiner, W.: Extending Floyd-Hoare logic for partial pre- and postconditions. In Ermolayev, V., Mayr, H., Nikitchenko, M., Spivakovsky, A., Zholtkevych, G., eds.: Information and Communication Technologies in Education, Research, and Industrial Applications. Volume 412 of Communications in Computer and Information Science. Springer International Publishing (2013) 355–378
9. Nikitchenko, M., Kryvolap, A.: Properties of inference systems for Floyd-Hoare logic with partial predicates. Acta Electrotechnica et Informatica **13** (2013) 70–78
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.: Information and Communication Technologies in Education, Research, and Industrial Applications. Volume 469 of Communications in Computer and Information Science. Springer International Publishing (2014) 117–138
11. Nikitchenko, M., Shkilniak, S.: Mathematical logic and theory of algorithms. Publishing house of Taras Shevchenko National University of Kyiv, Ukraine (in Ukrainian) (2008)
12. Nielson, H., Nielson, F.: Semantics with applications – a formal introduction. Wiley professional computing. Wiley (1992)
13. Wiedijk, F.: The seventeen provers of the world. Foreword by Dana S. Scott. Volume 3600 of Lecture Notes in Artificial Intelligence. Springer-Verlag Berlin Heidelberg (2006)
14. Gordon, M.: Mechanizing programming logics in higher order logic. In Birtwistle, G., Subrahmanyam, P., eds.: In Current Trends in Hardware Verification and Automated Theorem Proving. Springer-Verlag (1989)
15. Von Wright, J., Hekanaho, J., Luostarinen, P., Langbacka, T.: Mechanizing some advanced refinement concepts. Formal Methods in System Design (1993) 49–81
16. Bancerek, G., Byliński, C., Grabowski, A., Korniłowicz, A., Matuszewski, R., Naumowicz, A., Pąk, K., Urban, J.: Mizar: State-of-the-art and beyond. Volume 9150 of Lecture Notes in Computer Science. Springer (2015) 261–279
17. Grabowski, A., Korniłowicz, A., Naumowicz, A.: Four decades of Mizar. Journal of Automated Reasoning **55** (2015) 191–198
18. Nikitchenko, M., Tymofieiev, V.: Satisfiability in composition-nominative logics. Central Europ. J. Computer Science **2** (2012) 194–213
19. Dijkstra, E.: A Discipline of Programming. 1st edn. Prentice Hall PTR, Upper Saddle River, NJ, USA (1997)
20. Ivanov, I., Korniłowicz, A., Nikitchenko, M.: Formalization of nominative data in Mizar. Proceedings of TAAPSD 2015, 23-26 December 2015, Taras Shevchenko National University of Kyiv, Ukraine (2015) 82–85
21. Harel, D., Tiuryn, J., Kozen, D.: Dynamic Logic. MIT Press, Cambridge, MA, USA (2000)
22. Sannella, D., Tarlecki, A.: Foundations of Algebraic Specification and Formal Software Development. Monographs in Theoretical Computer Science. An EATCS Series. Springer (2012)