Completeness Theorems for Program-oriented Algebra-based Logics of Partial Quasiary Predicates

Mykola S. Nikitchenko and Stepan S. Sklyniak
Department of Theory and Technology of Programming
Taras Shevchenko National University of Kyiv
64, Volodymyrska Street, 01601 Kyiv, Ukraine
{nikitchenko, ssh}@unicyb.kiev.ua

Abstract. We provide motivation for developing and studying program-oriented logics of partial predicates. Such logics are algebra-based logics constructed in a semantic-syntactic style on the methodological basis that is common with programming; they can be considered as generalizations of traditional logics on classes of partial predicates that do not have fixed arity. Such predicates, called quasiary predicates, are defined over partial variable assignments (over partial data). We describe the hierarchy of different logics of quasiary predicates. For the constructed logics some laws of classical logic fail because of partiality of predicates and data. We construct sequent calculi for a number of defined logics and prove their soundness and completeness. The methods proposed can be useful for construction and investigation of logics for program reasoning.

Keywords: Partial predicate, partial logic, first-order logic, validity, sequent calculus, soundness, completeness.

1 Introduction

Mathematical logic is widely used for investigation of programs [1, 2]. Still, there are certain discrepancies between problems to be solved in this area and a logic in use. For example, such program properties as partiality of functions, elaborated system of data types, behavioural non-determinism etc. are difficult to investigate by traditional logic. To cope with such discrepancies we propose to construct logics based directly on program models.

To realize this idea we first construct models of programs using composition-nominative approach [3]. Principles of the approach (development of program notions from abstract to concrete, priority of semantics, compositionality of programs, and nominativity of program data) form a methodological base of program model construction. These principles specify program models as composition-nominative systems. Such a system may be considered as a triple of simpler systems: composition, description, and denotation systems. A composition system defines semantic aspects of programs, a description system defines program descriptions (syntactic aspects), and a denotation system specifies meanings (referents) of descriptions. We consider semantics of programs as partial functions over a class of data processed by programs; compositions are \( n \)-ary operations over functions. Thus, a composition system can be specified by two algebras: data algebra and function algebra. Function algebra is the main semantic notion in program formalization. Terms of this algebra define syntax of programs (description system), and ordinary procedure of term interpretation gives a denotation system.

The constructed formal program models form a base for developing of a rigorous mathematical formalism for reasoning about programs, in other words – a program logic. It is not possible to invent one universal program logic that would have all necessary properties. Therefore a hierarchy of logics, oriented on the hierarchy of program models, has to be developed. Here we briefly introduce such a hierarchy. Obtained logics are called composition-nominative logics (CNL).

It is important to admit that CNL better reflect program properties, but the opposite side of this feature is that the methods of logic investigation turn out to be more complicated. In this paper we continue our work on studying CNL focusing on the completeness problems for the first-order CNL.

The rest of the paper is structured as follows. In section 2 we give a motivational example of program formalization and logic construction. In section 3 we briefly introduce a hierarchy of logics of quasiary predicates. We define first-order pure CNL in section 4. In section 5 we consider properties of the consequence relation. In section 6 a sequent calculus for first-order pure CNL is defined; its soundness and completeness are demonstrated. In section 7 conclusions are formulated.

Proofs are omitted here and will be provided in an extended version of the paper.
2 Constructing Program Algebras and Logics: a Motivating Example

Let us consider the example language EL which is used here to demonstrate how program logics can be constructed. EL is similar to such languages as WHILE [4], IMP [5], etc.

The grammar of the language is defined as follows:

- \( s ::= x := a \mid s_1 ; s_2 \) if \( b \) then \( s_1 \) else \( s_2 \) while \( b \) do \( s \) end
- \( a ::= k \mid x \mid a_1 + a_2 \mid a_1 - a_2 \mid a_1 \cdot a_2 \mid a_1 \div a_2 \mid a_1 \mod a_2 \) (a)
- \( b ::= a_1 = a_2 \mid a_1 > a_2 \mid a_1 \lor a_2 \mid \neg (a) \)

where:

- \( k \) ranges over integers \( \{\ldots, -2, -1, 0, 1, 2, \ldots\} \),
- \( x \) ranges over variables (names) \( V = \{N, X, Y, Z, \ldots\} \)
- \( a \) ranges over arithmetic expressions \( Aexpr \),
- \( b \) ranges over Boolean expressions \( Bexpr \),
- \( s \) ranges over statements (programs) \( Stm \).

As an example consider an EL-program \( ES \) for calculating a function \( r = x^n \) (\( n \geq 0 \)) using Exponentiation by Squaring Algorithm. In this program variables \( N, X, \) and \( R \) denote integer values \( n, x, \) and \( r \) respectively:

\[
R := 1;
\]

while \( N > 0 \) do

\[
\text{if } (N \mod 2) = 1
\]

then begin \( R := R \times X; N := N - 1 \) end

else begin \( X := X \times X; N := N \div 2 \) end

Starting from this example we construct program algebras of three forms:

- first, we define semantics of \( ES \) in the style of denotational semantics; as a result we obtain a program algebra with \( n \)-ary mappings oriented on EL;
- then we represent \( n \)-ary mappings by quasiary mappings obtaining a simpler program algebra;
- at last, we define a general class of quasiary program algebras. This class of algebras is a semantic base for quasiary program logics. It captures main program properties that are invariant of such programs’ specifics as variable typing, interpreted predicates and functions, etc.

Analyzing the structure of the program we see that it is constructed from 1) symbols of \( n \)-ary operations (+, -, *, \( \div \), mod, =, \( > \)), 2) Boolean (\( N > 0 \), (\( N \mod 2 \) = 1) and arithmetic (1, 2, \( N - 1 \), \( N \div 2 \), \( N \mod 2 \), \( R \times X \), \( X \times X \)) expressions, and 3) statements obtained with the help of structuring constructs such as assignment, sequence, selection, and loop. Note that \( \div \) is partial on Int. To emphasize mapping’s partiality/totality we write the sign \( \longrightarrow^L \) for partial mappings and the sign \( \longrightarrow^T \) for total mappings.

We use denotational semantics (see, e.g. [4, 5]) to formalize the meaning of program components. Semantic mapping is represented as \( [\_] \). Three program components identified above determine three types of mappings called respectively \( n \)-ary, quasiary, and bi-quasiary mappings.

2.1 Classes of \( n \)-ary mappings

Symbols of arithmetic operations, relations, and Boolean connectives represent \( n \)-ary mappings defined on \( Int \) or on the set \( Bool = \{T, F\} \) of Boolean values.

For our language EL we define the following types of \( n \)-ary mappings:

\[
Fn^{n, Int} = Int^n \longrightarrow^p Int, Pr^{n, Int} = Int^n \longrightarrow^p Bool, Pr^{n, Bool} = Bool^n \longrightarrow^p Bool, n \geq 0.
\]

Using the same notation for language symbols of various types and mappings they represent, we can write that:

\[
+, -, \cdot, \div, \text{mod}; Fn^{2, Int}; =, >; Pr^{2, Int}; \lor; Pr^{2, Bool}; \Rightarrow; Pr^{1, Bool}.
\]
2.2 Classes of quasiary mappings

Quasiary mappings are defined over classes of states considered as sets of named values. For example, the expression \( R \times X \) specifies a function which given a state \( d \) of the form \( [r \mapsto r, X \mapsto x] \), where \( r \) and \( x \) are integers, evaluates a value \( r \cdot x \). Examples of states are \([X \mapsto 8, N \mapsto 4], [X \mapsto 8, N \mapsto 4, R \mapsto 8], [X \mapsto 8]\). In a state \( d \) a variable \( v \) can have a value (this is denoted \( d(v) \)) or be undefined (denoted \( d(v) \uparrow \)); thus, \([X \mapsto 8, N \mapsto 4][X] \downarrow \) and \([X \mapsto 8, N \mapsto 4][R] \uparrow \). Formally, the set of states \( \text{State} \) is defined as the set \( \forall a = a \rightarrow \text{Int} \) of all partial mappings from \( V \) to \( \text{Int} \).

Having described states we are able to represent formal semantics of Boolean and arithmetic expressions. Boolean expressions denote predicates (called partial quasiary predicates) of the set \( \text{Pr}^{\text{V,Int}} \rightarrow \text{Bool} \); thus for \( b \in \text{Bexpr} \) we have \([b] \in \text{Pr}^{\text{V,Int}}\). Integer expressions denote functions (called partial quasiary functions) of the set \( \text{Fn}^{\text{V,Int}} = \text{State} \rightarrow \text{Int} \); thus, for \( e \in \text{Aexpr} \) we have \([e] \in \text{Fn}^{\text{V,Int}}\). In the sequel we omit the term ‘partial’ for quasiary mappings. Since states are constructed with the help of naming (nominative) relation, they are also called nominative sets. Functions from \( \text{Fn}^{\text{V,Int}} \) are called ordinary functions since their ranges are sets of atomic (non-structured) values.

To represent semantics of variables in arithmetic expressions we will use a parametric denotation (denaming) functions \( \acute{x} : \text{Fn}^{\text{V,Int}} \). For a given program state, the function \( \acute{x} \) returns the value of the variable \( x \) in that state. For instance, denotation function that yields the value of name \( N \) is denoted by \( \acute{N} \). Such values may or may not be defined, so denotation functions are partial.

Semantics of integer numbers is treated as quasiary constant functions. Such functions are represented by constants \( k \) written in bold font; thus, \([1] = 1 \). It is clear that \( 1 \in \text{Fn}^{\text{V,Int}}\).

For specifying semantics of complex expressions special compositions called superpositions are used. Superposition \( S_F^n : \text{Fn}^{\text{V,Int}} \times \text{Fn}^{\text{V,Int}} \rightarrow \text{Fn}^{\text{V,Int}} \) of functions \( g_1, \ldots, g_n \) into an \( n \)-ary function \( f^n \) is an operator such that \( S_F^n (f^n, g_1, \ldots, g_n) (d) = f^n (g_1(d), \ldots, g_n(d)) \) where \( d \) is a state. The same formula can be used for defining superposition \( S_B^n : \text{Pr}^{\text{V,Int}} \times \text{Fn}^{\text{V,Int}} \rightarrow \text{Pr}^{\text{V,Int}} \) of functions \( g_1, \ldots, g_n \) into \( n \)-ary predicate \( p^n \), and for superposition \( S_B^0 : \text{Pr}^{\text{Bool}} = \text{Pr}^{\text{V,Int}} \rightarrow \text{Pr}^{\text{V,Int}} \) of predicates \( p_1, \ldots, p_n \) into an \( n \)-ary Boolean function \( f^n_B \). Thus, \([N \land 1] = S_F^2 (\land, '\land', 1), \quad [N \land 0] = S_F^2 (\lor, '\lor', 0), \quad [(N \mod 2) = 1] = S_F^2 (\equiv, '(\mod, 'N', 2)', 1)\). For instance, \( [R := R \times X; N := N - 1] = \text{AS}^R (S_B^2 (\alpha, 'R', 'X') \bullet \text{AS}^N (S_F^2 (\land, '\land', 'N', 1)))\).

Note, that we define \( \bullet \) by commuting arguments of conventional functional composition: \( f \circ g = g \circ f \).

2.3 Classes of bi-quasiary mappings

Programs (statements) from \( \text{Stm} \) denote bi-quasiary functions (program functions) of the class \( \text{FPrg}^{\text{V,Int}} = \text{State} \rightarrow \text{State} = \forall \text{Int} \rightarrow \forall \text{Int} \). Such functions are also called bi-nominative.

Semantics of structured statements is defined by the following compositions with conventional meaning:

1. assignment composition \( \text{AS}^\times : \text{Fn}^{\text{V,Int}} \rightarrow \text{FPrg}^{\text{V,Int}} \) (\( x \) is a parameter from \( V \));
2. composition of sequential execution \( \bullet : \text{FPrg}^{\text{V,Int}} \times \text{FPrg}^{\text{V,Int}} \rightarrow \text{FPrg}^{\text{V,Int}} \);
3. conditional composition \( \text{IF} : \text{Pr}^{\text{V,Int}} \times \text{FPrg}^{\text{V,Int}} \times \text{FPrg}^{\text{V,Int}} \rightarrow \text{FPrg}^{\text{V,Int}} \);
4. loop composition \( \text{WH} : \text{Pr}^{\text{V,Int}} \times \text{FPrg}^{\text{V,Int}} \rightarrow \text{FPrg}^{\text{V,Int}} \).

For instance, \( [R := R \times X; N := N - 1] = \text{AS}^R (S_B^2 (\alpha, 'R', 'X') \bullet \text{AS}^N (S_F^2 (\land, '\land', 'N', 1)))\).

Note, that we define \( \bullet \) by commuting arguments of conventional functional composition: \( f \circ g = g \circ f \).

2.4 Program algebra with \( n \)-ary mappings

The definitions introduced permit to conclude that the following program algebra with \( n \)-ary mappings oriented on ET has been constructed (we omit types of compositions):

\[
\text{APn} (V, \text{Int}) = \langle \text{Fn}^{2,\text{Int}}, \text{Pr}^{2,\text{Bool}}, \text{Pr}^{1,\text{Bool}}, \text{Pr}^{1,\text{Int}}, \text{Pr}^{1,\text{Int}}, \text{FPrg}^{\text{V,Int}}; +, -, \cdot, \div, \text{mod}, =, \geq, \leq, \geq, \vdash, k, S_F^2, S_B^2, S_B^1, \cdot, x, \text{AS}^\times, \bullet, \text{IF}, \text{WH} \rangle.
\]
Note that notation for parametric compositions (like denominations, assignments etc.) represents here classes of compositions. Thus, ‘r’ represents the class of compositions for various x; k represents the class of integer constants treated as quasiary constant functions.

We would like to emphasize the fact that semantics of EL-programs (or EL-expressions) can be represented as terms of this algebra. This simplifies investigations of EL-programs because the constructed algebra completely specifies their semantics.

The term for EL-program $ES$ is as follows:

$$AS^n(1) \bullet WH(S^n_F(\setminus, 'N', 0)), IF(S^n_F(=, S^n_F \mod ('N', 2), 1),$$

$$AS^n(S^n_F(\setminus, 'R', 'X')) \bullet AS^n(S^n_F(\setminus, 'N', 1)), AS^n(S^n_F(\setminus, 'X', 'X')) \bullet AS^n(S^n_F(\setminus, 'N', 2))).$$

Note that this term and its sub-terms can denote partial mappings, as the function $\div$ and denomination functions can be undefined; also $WH$ composition can be a source of undefinedness.

Having specified this algebra we can study properties of programs; this can be used in program reasoning. For example, it is possible to prove commutativity of the assignment statements $R := R \cdot X$ and $N := N - 1$ by proving in the algebra $APn(V, Int)$ the corresponding property of composition of sequential execution:

$$AS^n(S^n_F(\setminus, 'R', 'X')) \bullet AS^n(S^n_F(\setminus, 'N', 1)) = AS^n(S^n_F(\setminus, 'N', 1)) \bullet AS^n(S^n_F(\setminus, 'R', 'X)).$$

We can also prove more general properties; say we can prove associativity of sequential execution of statements:

$$\begin{align*}
\text{begin } s_1 : s_2 \text{ end} & = s_1 : \text{begin } s_2 : s_3 \text{ end} \\
\text{begin } b \text{ then begin } s_1 \text{ else begin } s_2 \text{ end } \text{ else begin } s_2 : s_3 \text{ end}.
\end{align*}$$

Its validity is based on the property of $APn(V, Int)$ that $IF(p, f, g) \bullet h = IF(p, f \bullet h, g \bullet h). (Here f, g, h \in Fn^{V,Int}, p \in Pr^{V,Int}).$

Still, the constructed program algebra with n-ary mappings looks overcomplicated; therefore we construct a simpler algebra without n-ary mappings. It is possible because n-ary mappings can be mimicked by quasiary mappings. Still, we should first split the set of operation symbols of this algebra into two parts: logical symbols, which have relatively type-independent interpretations, and non-logical (or descriptive) symbols, which represent specifics of the carriers. Logical symbols will be treated as compositions and descriptive symbols as quasiary mappings. Trivial inspection of definitions shows that symbols $\setminus, \bullet, \star, \div, \mod, =, >$ are descriptive symbols in the signature of $APn(V, Int)$. They are defined over integer numbers, and may be considered constants in this algebra. Other symbols may be considered logical.

### 2.5 Program algebra without n-ary mappings

The considerations described above open the second phase of program algebra development. To make the algebra simpler we can exclude from it the classes of n-ary functions and predicates, thus concentrating on logical symbols that are interpreted as compositions over nominative carriers $Pr^{V,Int}, Fn^{V,Int}$, and $FP\setminus V,Int$. Still, n-ary functions and predicates can be represented in these classes of nominative mappings.

We explain the idea of representation of n-ary mappings on the example of binary multiplication function. First, we represent a pair $(x_1, x_2)$ as a state $[1 \mapsto x_1, 2 \mapsto x_2]$, where 1 and 2 should be treated as standard variables that represent the arguments of a binary function symbol. This permits to treat multiplication as a quasiary function. Then, in order to avoid usage of standard names 1 and 2 and to obtain homogeneity of names we can introduce a parametric quasiary function $x \cdot y$ (printed in bold font) such that $x \cdot y = S^2_F (\bullet, x', y)$; here $x$ and $y$ are parameters from $V$.

Therefore instead of binary functions we introduce parametric quasiary functions $x+y, x-y, x \cdot y$, $x \div y$, and $x \mod y$ over $Int$; also instead of relations we introduce new parametric quasiary predicates $x=y$ and $x \neq y$ (with $x$ and $y$ as parameters).

This step permits to represent every n-ary function defined over $Int$ as a parametric quasiary function. But now, to represent the semantics of complex expressions we should introduce special superpositions $S^n_F(\setminus, \cdot, \bullet, \star, \div, \mod)$, and $S^n_F(\setminus, \cdot, \bullet, \star, \div, \mod)$, which are called superpositions into quasiary function and predicate respectively: $S^n_F(\setminus, \cdot, \bullet, \star, \div, \mod)$, and $S^n_F(\setminus, \cdot, \bullet, \star, \div, \mod)$.
Intuitive meaning of these formulas is that we change in \( d \) the values of names \( v_1, \ldots, v_n \) to \( g_i(d), \ldots, g_n(d) \) respectively (partiality should be taken into account). Thus, semantics, say, of the expression \( R+(R \cdot X) \), can be represented as \( S^V_P \ (R+Y, R \cdot X) \).

Now let us consider logical symbols \( \vee: \text{Bool}^2 \longrightarrow \text{Bool} \) and \( \neg: \text{Bool} \longrightarrow \text{Bool} \). We cannot directly represent them as quasiary predicates, therefore we advocate another approach. We will treat them as binary compositions over quasiary predicates (denoted by the same signs) \( \vee: \text{Pr}^{V_{\text{Int}}} \times \text{Pr}^{V_{\text{Int}}} \longrightarrow \text{Pr}^{V_{\text{Int}}} \) and \( \neg: \text{Pr}^{V_{\text{Int}}} \longrightarrow \text{Pr}^{V_{\text{Int}}} \). Such representations also provide better possibilities to work with partial predicates. For example, consider a Boolean expression \( (R \cdot X) \vee (R > X) \). Its semantics in \( \text{APn}(V, \text{Int}) \) is represented by the term \( S^V_{R} (\vee, S^V_{X} (\neg, S^V_{R} (\neg, X)), R), S^V_{R} (\neg, R, X) \). But superposition into an \( n \)-ary mapping is strict: when one argument is not defined then the result is also undefined. This property restricts possibilities of construction of new partial predicates. For example, for Kleene’s strong disjunction [6] it is allowed that one argument may be undefined if the other one is evaluated to true. When we represent connectives as compositions, we avoid the above considered difficulties.

Thus, we can now consider a simpler algebra – the program algebra of quasiary predicates with constants (without \( n \)-ary mappings):

\[
\text{APQC}(V, \text{Int}) = \langle \text{Pr}^{V_{\text{Int}}}, \text{Fn}^{V_{\text{Int}}}, \text{FPg}^{V_{\text{Int}}};
\quad x-y, x \cdot y, x \div y, x \mod y, x \cdot y, \neg, \emptyset, \text{AS}^V, \bullet, \text{IF}, \text{WH} \rangle.
\]

In this algebra we have 8 parametric descriptive symbols considered as algebra’s constants, and 9 logical symbols. Note, that all logical symbols are treated as compositions, possibly null-ary compositions (as in the case of denomination functions).

Semantics of \( \text{ES} \) program is represented by the following term of this algebra:

\[
\text{AS}^V(1) \bullet \text{WH}(N \cdot 0) \cdot \text{IF}((N \mod 2) = 1) \cdot \text{AS}^V(N-1) \cdot \text{AS}^V(X \cdot X) \cdot \text{AS}^V(N \div 2)).
\]

### 2.6 Program algebra without descriptive constants

The next step of constructing more “logical” algebras consists of eliminating from \( \text{APQC}(V, \text{Int}) \) descriptive symbols \( x \cdot y, x-y, x \div y, x \mod y, x \cdot y \) with fixed interpretations. We obtain a program algebra of quasiary predicates \( \text{APQC}(V, \text{Int}) \) without constants:

\[
\text{APQC}(V, \text{Int}) = \langle \text{Pr}^{V_{\text{Int}}}, \text{Fn}^{V_{\text{Int}}}, \text{FPg}^{V_{\text{Int}}};
\quad \vee, \neg, S^V_{F}, S^V_{P}, \emptyset, \text{AS}^V, \bullet, \text{IF}, \text{WH} \rangle.
\]

As to eliminated descriptive symbols, we can instead consider sets \( \text{Ps}, \text{Fs}, \text{Fps} \) of predicate, ordinary function, and program function symbols that do not have predefined interpretations, and consequently, can denote any quasiary predicate or function or bi-quasiary function.

The algebra \( \text{APQC}(V, \text{Int}) \) is based on integer numbers, though it is clear that we can consider other data types, say \( \text{real}, \text{nat}, \text{etc.} \). Also, new operations over such data types can be considered. So, one can ask a question: what EL-program properties remain valid under type and operation variations?

### 2.7 General program algebras (with arbitrary classes of basic values)

Being interested in general laws of reasoning about programs we should make the next step and define compositions for any set \( A \) of basic (atomic) values. In this case we obtain the following program algebra of quasiary mappings:

\[
\text{APQC}(V, A) = \langle \text{Pr}^{V_A}, \text{Fn}^{V_A}, \text{FPg}^{V_A};
\quad \vee, \neg, S^V_{F}, S^V_{P}, \emptyset, \text{AS}^V, \bullet, \text{IF}, \text{WH} \rangle.
\]

Symbols from \( \text{Ps}, \text{Fs}, \text{Fps} \) are used to construct terms of this algebra. Properties of such terms are general properties because they should be valid under any interpretations of function and predicate symbols.

It means that we have constructed a class of quasiary program algebras (for various \( A \)), representing program semantics for languages with different domains. Such algebras may be called general program models; they form the semantic base for program logics.

For example, we can consider equational program logics by defining formulas of these logics as formal equalities of the form \( t_1 = t_2 \), where \( t_1 \) and \( t_2 \) are terms of the type \( \text{FPg}^{V_A} \). Such logics define equivalent transformations of programs.

Another conventional program logic is Floyd–Hoare logic, which is based on assertions of the form \( \{b_1\} s \{b_2\} \). Semantics of such assertions can be represented by Floyd–Hoare composition \( \text{FH}: \text{Pr}^{V_A} \times \text{FPg}^{V_A} \longrightarrow \text{Pr}^{V_A} \). We define this composition under assumption that predicates and functions can be partial [7]. Then
Extending the algebra $APQ(V, A)$ with $FH$ composition, unary parametric compositions of renomination $R^x_V$, existential quantification $\exists x$ (to be defined in the next sections), and a composition of equality $=$ we obtain a three-sorted algebra $APQFH(V, A) = <$ $Pr^V_A$, $Fn^V_A$, $FPrg^V_A$, $\lor$, $\neg$, $R^x_V$, $S^F_F$, $S^P_P$, $\exists x$, $\exists$, $=, AS^x$, $\bullet$, $IF$, $WH$, $FH$ $>$ presented in Fig. 1.

Figure 1. Three-sorted algebra of quasiary predicates, ordinary and program functions.

2.8 General program algebras as a semantic base of program logics

The class of algebras $APQFH(V, A)$ forms the semantic base for quite powerful Floyd–Hoare-like logics of quasiary mappings. It is important to admit that by restricting this algebra on one and two carriers we obtain respectively the following two algebras:

- propositional algebra $AP(V, A) = <$ $Pr^V_A$, $\lor$, $\neg$, $>$;
- first-order algebra of quasiary predicates and ordinary functions $AQFO(V, A) = <$ $Pr^V_A$, $Fn^V_A$, $\lor$, $\neg$, $R^x_V$, $S^F_F$, $S^P_P$, $\exists x$, $\exists$, $=, AS^x$ $>$.

Based on classes of such algebras various propositional, first-order, and program CNL can be defined (see the next sections). We can go further and define modal and temporal CNL [8].

All logics are constructed in a semantic-syntactic style: first, semantic component of a logic is defined as a class of certain algebras, then syntactic component (a logic language) is described, and, at last, interpretational component is specified.

2.9 Special features of quasiary predicates and functions

Before continuing with formal definitions it is desirable to give an intuitive understanding of logics of quasiary predicates and to compare them with classical logic of $n$-ary predicates.
We identify the following properties of quasiary predicates:

- partiality of predicates;
- unrestricted (possibly infinite) arity of predicates;
- sensitivity of predicates to unassigned variables (sensitivity to partiality of data).

These features complicate investigation of logics of quasiary predicates comparing with classical logic and violate some laws of classical logic.

In particular, partiality of predicates violates Modus Ponens [9]. Indeed, let und be interpreted as a nowhere defined predicate and $\Phi$ be interpreted as a refutable predicate. We treat validity as irrefutability, therefore $\text{und}$ and $\text{und} \rightarrow \Phi$ are valid (irrefutable), but $\Phi$ is refutable.

Unrestricted arity of quasiary predicates violates the property that a sentence (a closed formula) has a constant value in a fixed interpretation because a predicate, obtained by formula interpretation, can depend upon variable values that do not occur in that formula.

Sensitivity also affects the logic laws. Let us illustrate this with a simple example. Define a parametric predicate $\varepsilon \in \mathcal{P}r_{V,A}$ ($\varepsilon \in V$) by the following formulas: $\varepsilon(z)(d) \triangleq F$ if $z$ is assigned in $d$ and $\varepsilon(z)(d) \triangleq T$ if $z$ is unassigned. This quasiary predicate is called variable unassignment predicate. It has different values depending whether a value of $z$ is assigned or unassigned; thus, it is sensitive to unassigned variables. Consider the following phrase as one of intuitive interpretations of a predicate $\varepsilon$:

Thus, this phrase evaluates to $T$ in situations (states) when food is not assigned; when this variable is assigned then the phrase evaluates to $F$. The formula $(\forall \varepsilon; (\varepsilon(\text{food}) \rightarrow \neg \varepsilon(\text{food})) \rightarrow \neg \varepsilon(\text{food})$ may fail in situations (states) when my cat does not have any food. In Tarski’s semantics of classical logic we have double totality: predicates are total and variable assignments (data) are total. Therefore in classical logic the formula $(\forall \varepsilon; (\varepsilon(\text{food}) \rightarrow \neg \varepsilon(\text{food})) \rightarrow \neg \varepsilon(\text{food})$ is always valid. Such sensitivity complicates logic calculus therefore more powerful instruments should be introduced. As such instruments we will use an infinite set of unessential variables and variable unassignment predicates (see the next sections).

Summing up, we would like to say that semantics of programs can be presented by terms of program algebras with compositions as operations in these algebras; program functions, ordinary functions, and predicates are quasiary mappings defined on nominative sets (nominative data); we define program logics directly on program algebras by extending their signatures with special “logical” compositions.

3 Hierarchy of Composition-Nominative Logics

Three kinds of logics can be constructed from program models over nominative sets (see Fig. 1):

1) pure quasiary predicate logics based on algebras with one sort: $\mathcal{P}r_{V,A}$
2) quasiary predicate-function logics based on algebras with two sorts: $\mathcal{P}r_{V,A}$ and $\mathcal{F}n_{V,A}$
3) quasiary program logics based on algebras with three sorts: $\mathcal{P}r_{V,A}$, $\mathcal{F}n_{V,A}$, and $\mathcal{F}Prg_{V,A}$.

For logics of pure quasiary predicates (pure CNL) we identify renominative, quantifier, and quantifier-equational levels.

Renominative logics [8] are the most abstract among above-mentioned logics. The main new compositions for these logics are the compositions of renomination (renaming) of the form $R_{\lambda_{1} \ldots \lambda_{n}}^{\eta_{1} \ldots \eta_{n}} : \mathcal{P}r_{V,A} \longrightarrow \mathcal{P}r_{V,A}$. Intuitively, given a quasiary predicate $p$ and a nominative set $d$ the value of $R_{\lambda_{1} \ldots \lambda_{n}}^{\eta_{1} \ldots \eta_{n}}(p)(d)$ is evaluated in the following way: first, a new nominative set $d'$ is constructed from $d$ by changing the names $v_{1}, \ldots, v_{n}$ in $d$ to the values of the names $x_{1}, \ldots, x_{n}$ respectively; then the predicate $p$ is applied to $d'$. The obtained value (if it was evaluated) will be the result of $R_{\lambda_{1} \ldots \lambda_{n}}^{\eta_{1} \ldots \eta_{n}}(p)(d)$. For this composition we will also use a simplified notation $R_{\lambda}^{\eta}$. The basic compositions of renominative logics are $\vee$, $\neg$, and $R_{\lambda}^{\eta}$.

Note, that renomination (primarily in syntactical aspects) is widely used in classical logic, lambda-calculus, and specification languages like Z-notation [10], B [11], TLA [12], RAISE [13], ASM [14] etc.

At the quantifier level, all basic values can be used to construct different nominative sets to which quasiary predicates can be applied. This allows one to introduce the compositions of quantification of the form $\exists x$ in style of Kleene’s strong quantifiers. The basic compositions of logics of the quantifier level are $\vee$, $\neg$, $R_{\lambda}^{\eta}$, $\exists x$.

At the quantifier-equational level, new possibilities arise for equating and differentiating values with special 0-ary compositions of the form $\equiv_{0}$ called equality predicates. Basic compositions of logics of the quantifier-equational level are $\vee$, $\neg$, $R_{\lambda}^{\eta}$, $\exists x$, and $\equiv_{0}$.

All specified logics (renominative, quantifier, and quantifier-equational) are based on algebras that have only one sort: a class of quasiary predicates.
For quasiary predicate-function logics we identify the function level and the function-equational level.

At the function level, we have extended capabilities of formation of new arguments for functions and predicates. In this case it is possible to introduce the superposition compositions \( S_F^p \) and \( S_P^p \) (see [8]), which formalize substitution of functions into function and predicate respectively. Also special 0-ary denomination parametric compositions \( \lambda \) are introduced. Introduction of such functions allows one to model renomination compositions with the help of superpositions. The basic compositions of logics of the function level are \( \vee, \neg, S_F^p, S_P^p, \exists x, \lambda \).

At the function-equational level, a special equality composition \( = \) can be introduced additionally [8]. The basic compositions of logics of the function-equational level are \( \vee, \neg, S_F^p, S_P^p, \exists x, \lambda, = \). At this level different classes of first-order logics can be presented.

This means that two-sorted algebras (with sets of predicates and functions as sorts and above-level mentioned compositions as operations) form a semantic base for first-order CNL.

To preserve properties of classical first-order logic in first-order CNL we should restrict the class \( \forall A \longrightarrow \text{Bool} \) of quasiary predicates. Namely, we introduce a class of equitone predicates and its different variations such as maxitotal equitone predicates, equicompatible predicates, etc. [8]. A predicate \( p: \forall A \longrightarrow \text{Bool} \) is called equitone if for every \( d \in \forall A \) such that \( d \equiv d' \) from \( p(d) \equiv b \) follows that \( p(d') \equiv b \); if an equitone predicate \( p \) is defined on all elements of \( A \) then \( p \) is said to be maxitotal equitone; if a predicate \( p \) is a restriction of some equitone predicate then \( p \) is equicompatible predicate. Here \( A \) denotes all total mappings from \( V \) to \( A \) (total assignments). Logics based on maxitotal equitone, equitone, and equicompatible predicates are the “closest” generalizations of the classical first-order logic that preserve its main properties. These logics are called neoclassical logics [8].

The level of program logics is quite rich. Investigation of such logics is a special challenge; here we only mention Floyd-Hoare-type logic based on a monotone Floyd-Hoare composition [7].

In the rest of the paper we consider a first-order composition-nominative pure quasiary predicate logic denoted \( L^Q \). Such logic plays a central role in logics hierarchy and is based on algebras \( AQ(V, A) = \langle Pr^{V,A}, \ empowering predicates \rangle \) for various \( A \). We will construct a sequent calculus for this logic and prove its soundness and completeness.

### 4 Formal Definitions of First-Order Composition-Nominative Pure Quasiary Predicate Logic

To define the logic \( L^Q \) we have to specify its semantic, syntactic, and interpretational components [15, 16].

#### 4.1 Semantic component

Let \( V \) be a set of names. According to tradition, names from \( V \) are also called variables. Let \( A \) be a set of basic values. Given \( V \) and \( A \), the class \( \forall A \) of nominative sets is defined as the class of all partial mappings from \( V \) to \( A \), thus, \( \forall A \simeq V \longrightarrow A \). Informally speaking, nominative sets represent states of variables.

Though nominative sets are defined as mappings, we follow mathematical traditions and also use set-like notation for these objects. In particular, the notation \( d = \{ v_i \in a_i | i \in I \} \) describes a nominative set \( d \); the notation \( v_i \mapsto a_i \in d \) means that \( d(v_i) \) is defined and its value is \( a_i \). The main operation for nominative sets is a total unary parametric renomination \( r^{v_1, \ldots, v_n}_{x_1, \ldots, x_n}: \forall A \longrightarrow \forall A \), where \( v_1, \ldots, v_n, x_1, \ldots, x_n \in V, v_1, \ldots, v_n \) are distinct names, \( n \geq 0 \), which is defined by the following formula: \( r^{v_1, \ldots, v_n}_{x_1, \ldots, x_n}(d) = \{ v \mapsto a \in_a d | \forall v \in \{ v_1, \ldots, v_n \} \cup \{ v_i \mapsto d(x_i) | d(x_i) \downarrow, i \in \{ 1, \ldots, n \} \} \} \). Intuitively, given \( d \) this operation yields a new nominative set changing the values of \( v_1, \ldots, v_n \) on the values of \( x_1, \ldots, x_n \) respectively. The set of assigned names in \( d \) is defined by the formula \( \text{asn}(d) = \{ \forall v \in V | v \mapsto a \in_a d \) for some \( a \in A \} \).

Let \( \text{Bool} = \{ F, T \} \) be a set of Boolean values. Let \( Pr^{V,A} \vdash \forall A \longrightarrow \text{Bool} \) be a set of all partial predicates over \( \forall A \). Such predicates are called partial quasiary predicates. The term “partial” is usually omitted.

For \( p \in Pr^{V,A}, v \in V, a \in A \) we write:

- \( p(d) \downarrow \) to denote that \( p \) is defined on a nominative set \( d \);
- \( p(d) \downarrow = b \) to denote that \( p \) is defined on \( d \) with a Boolean value \( b \);
- \( p(d) \uparrow \) to denote that \( p \) is undefined on \( d \);
we inductively define the language of $\in \{ \in \}$ can represent one composition, when $\in \in$ and a set $\in \in \in$. Thus, notation $\in -3$.

The truth and falsity domains of $p$ are respectively $T(p) = \{d \in V | p(d) = T\}$ and $F(p) = \{d \in V | p(d) = F\}$. A predicate $p$ is irrefutable, or partially valid, if $F(p) \neq \emptyset$.

Types Operations over $Pr^V.A$ are called compositions. For $L^Q$ the set $C(V)$ of compositions is $\{ \lor, \land, R^T, \exists x \}$. Compositions have the following: $\lor: Pr^V.A \times Pr^V.A \stackrel{I}{\rightarrow} Pr^V.A$; $\land, R^i_{x_1,\ldots,x_n}$; $\exists x: Pr^V.A \stackrel{I}{\rightarrow} Pr^V.A$ and are defined by the following formulas $(p, q \in Pr^V.A)$:

- $T(p \lor q) = T(p) \cup T(q)$; $F(p \lor q) = F(p) \cap F(q)$;
- $T(\neg p) = T(p)$; $F(\neg p) = F(p)$;
- $T(R^T_x (p)) = r^T_x (T(p))$; $F(R^T_x (p)) = r^T_x (F(p))$;
- $T(\exists x p) = \{d \in V | p(d \forall x \rightarrow a) = T$ for some $a \in A\}$; $F(\exists x p) = \{d \in V | p(d \forall x \rightarrow a) = F$ for every $a \in A\}$.

Here $d \forall x \rightarrow a = \{v \rightarrow c \in d \mid v \neq x\} \cup \{x \rightarrow a\}$.

Note that parametric compositions of existential quantification and renomination can also represent classes of compositions. Thus, notations $\forall x$ can represent one composition, when $x$ is fixed, or a class $\{ \forall x \mid x \in V \}$ of such compositions for various names. Also note that we treat a parameter $v_1,\ldots,v_n$ of renomination composition as a total mapping from $\{ v_1, \ldots, v_n \}$ into $\{ x_1, \ldots, x_n \}$ thus parameters obtained by pairs permutations are identical.

A pair $AQ(V, A) = \langle Pr^V.A, C(V) \rangle$ is called a first-order algebra of quasiary predicates. Such algebras form a semantic base for the constructed first-order composition-nominative pure quasiary predicate logic $L^Q$. Let us now proceed with syntactic and interpretational components of this logic.

### 4.2 Syntactic component

A syntactic component specifies the language of $L^Q$. Let $Cs(V)$ be a set of composition symbols that represent compositions in algebras defined above, $Cs(V) = \{ \lor, \land, R^T, \exists x \}$. For simplicity, we use the same notation for symbols of compositions and compositions themselves.

Let $Ps$ be a set of predicate symbols. A triple $\Sigma^Q = (V, Cs(V), Ps)$ is a signature of a language of $L^Q$. Given a language signature $\Sigma^Q$, we inductively define the language of $L^Q$ – the set of formulas $Fr(\Sigma^Q)$:

1. if $P \in Ps$ then $P \in Fr(\Sigma^Q)$;
2. if $\Phi, \Psi \in Fr(\Sigma^Q)$ then $(\Phi \lor \Psi) \in Fr(\Sigma^Q)$;
3. if $\Phi \in Fr(\Sigma^Q)$ then $\neg \Phi \in Fr(\Sigma^Q)$;
4. if $\Phi \in Fr(\Sigma^Q)$, $v_1, \ldots, v_n$, $x_1, \ldots, x_n \in V$, $v_1, \ldots, v_n$ are distinct names, $n \geq 0$, then $R^i_{x_1,\ldots,x_n} \Phi \in Fr(\Sigma^Q)$;
5. if $\Phi \in Fr(\Sigma^Q)$, $x \in V$ then $\exists x \Phi \in Fr(\Sigma^Q)$.

### 4.3 Interpretational component

Given $\Sigma^Q$ and a set $A$ we can define an algebra of quasiary predicates $AQ(V, A) = \langle Pr^V.A, C(V) \rangle$. Composition symbols have fixed interpretation, but we additionally need interpretation $I^{Pr^V.A}: Ps \longrightarrow Pr^V.A$ of predicate symbols to obtain a language interpretation. A corresponding tuple $J = (\Sigma^Q, I^{Ps})$ is called an $L^Q$-interpretation.

Given a formula $\Phi$ and an $L^Q$-interpretation $J$ we can speak of an interpretation of $\Phi$ in $J$. It is denoted by $\Phi_J$.

For the logic $L^Q$ derived compositions (such as conjunction &; universal quantification $\forall x$, etc.) are defined in a traditional way.
Formulas and interpretations in \( L^0 \) are called \( L^0 \) -formulas and \( L^0 \) -interpretations respectively. Usually the prefix \( L^0 \) is omitted. A formula \( \Phi \) is called valid in interpretation \( J \) if there is no nominative set \( d \in \mathcal{A} \) such that \( \Phi|_d \models F \). This is denoted \( J \models \Phi \), which means that \( \Phi \) is not refutable in \( J \). A formula \( \Phi \) is called valid if \( J \models \Phi \) for every interpretation \( J \). We shall denote this \( |\models_{L^0} \Phi \), or just \( \models \Phi \) if the logic in hand is understood from the context.

4.4 Extensions of \( L^0 \)

The logic \( L^0 \) being a rather powerful logic still is not expressible enough to represent transformations required for proving its completeness. Therefore we introduce its two extensions: \( L^U \) – a logic with unessential variables, and \( L^U_d \) – a logic with unessential variables and a parametric variable unassignment predicate \( \varepsilon \) which checks whether a variable \( z \) is unassigned in a given nominative set.

To define \( L^U \) we should specify its semantic, syntactic, and interpretational components.

Let \( U \) be an infinite set of variables such that \( V \cap U = \emptyset \). Variables from \( U \) are called unessential variables (analog of fresh variables in classical logic) that should not affect the formula meanings [15]. Algebras \( AQ(V \cup U, A) = < P_d^{V \cup U, A}, C(V)> \) (for different \( A \) ) form a semantic base for \( L^U \). A syntactic component is specified by the set of formulas \( Fr(\Sigma^U) \) where \( \Sigma^U = (V \cup U, Cx(V \cup U), Ps) \) is the signature of \( L^U \). An interpretational component of \( L^U \) restricts the class of \( L^U \)-interpretations in such a way that interpretations of predicate symbols are neither sensitive to the values of the component nor to presence of such components. Formally, a variable \( u \in U \) is unessential in an interpretation \( Fr(\Sigma^U) \) if \( Fr(\Sigma^U)(d) = Fr(\Sigma^U)(d \uparrow u \rightarrow a) \) for all \( P \in Ps \), \( d \in V \cup A \), \( a \in A \).

The logic \( L^U_d \) is an extension of \( L^U \) by a null-ary parametric composition (predicate) \( \varepsilon \) \(( z \in V \cup U \) defined by the formulas: \( T(\varepsilon z) = (A(d(z)) = \{d \uparrow A | z \in as(d)\} \) and \( F(\varepsilon z) = (A(d(z)) = \{d \downarrow A | z \in as(d)\} \). Thus, for this logic the set of compositions is equal to \( \{\lor, \land, R_{\varepsilon}^a, \exists x, \varepsilon z\} \). Algebras of the form \( AQ(V \cup U, A) = < P_d^{V \cup U, A}, \lor, \land, R_{\varepsilon}^a, \exists x, \varepsilon z> \) constitute a semantic base for \( L^U_d \). A syntactic component is specified by the set of formulas \( Fr(\Sigma^U_d) \) where \( \Sigma^U_d = (V \cup U, \lor, \land, R_{\varepsilon}^a, \exists x, \varepsilon z), Ps) \) is the signature of \( L^U_d \). An interpretational component of \( L^U_d \) is defined in the same way as for \( L^U \).

Predicates \( \varepsilon \) specify cases when \( z \) is assigned or unassigned. This property can be used for quantifier elimination that is semantically supported for every algebra \( AQ(V \cup U, A) \) by the following statement:

\[
T(R_{\varepsilon}^a \Phi(d)) \cap F(\varepsilon y) \subseteq T(R_{\varepsilon}^a \Phi(d \uparrow a)) \text{ and } F(R_{\varepsilon}^a \Phi(d \downarrow a)) \subseteq F(R_{\varepsilon}^a \Phi(d)).
\]

Proof. Let \( d \in T(R_{\varepsilon}^a \Phi(d \uparrow a)) \cap F(\varepsilon y) \), then \( d(y) \downarrow a \) and \( R_{\varepsilon}^a \Phi(d \downarrow a) = T \), therefore \( d(y) \downarrow a \) for some \( a \in A \) and \( P(d \uparrow \pi) \rightarrow d(\pi) \downarrow x \rightarrow d(y) \downarrow a \) = \( T \). Hence, \( P(d \uparrow \pi) \rightarrow d(\pi) \downarrow x \rightarrow d(y) \downarrow a \) = \( T \) for some \( a \in A \). Thus, \( (\exists x P)(d) = T \), therefore \( R_{\varepsilon}^a (\exists x P)(d) = T \), this means that \( d \in T(R_{\varepsilon}^a (\exists x P)) \). So, \( T(R_{\varepsilon}^a \Phi(d \uparrow a)) \cap F(\varepsilon y) \subseteq T(R_{\varepsilon}^a (\exists x P)) \).

Let \( d \in F(R_{\varepsilon}^a (\exists x P)) \), then \( d(y) \downarrow a \) and \( R_{\varepsilon}^a (\exists x P)(d) = F \). From this \( (\exists x P)(d \uparrow \pi) \rightarrow d(\pi) \downarrow x \rightarrow b \Rightarrow F \) for all \( b \in A \). Since \( d(y) \downarrow a \) we have \( d(y) \downarrow a \) for some \( a \in A \), then \( P(d \uparrow \pi) \rightarrow d(\pi) \downarrow x \rightarrow d(y) \downarrow a \) = \( F \). From this \( R_{\varepsilon}^a (\exists x P)(d) = F \), which gives \( d \in F(R_{\varepsilon}^a (\exists x P)) \). So, \( F(R_{\varepsilon}^a (\exists x P)) \cap F(\varepsilon y) \subseteq F(R_{\varepsilon}^a (\exists x P)) \).

As a special case we get \( T(R_{\varepsilon}^a (\exists x P)) \cap F(\varepsilon y) \subseteq T(\exists x P) \cap F(\exists x P) \cap F(\varepsilon y) \subseteq F(R_{\varepsilon}^a (\exists x P)) \).

5 Consequence Relation for Sets of Formulas

Consequence relation is defined in the same way for all logics under consideration. So, we present its definition only for a logic \( L^U_d \).

Let \( \Gamma \subseteq Fr(\Sigma^U_d) \) and \( \Delta \subseteq Fr(\Sigma^U_d) \) be sets of formulas. \( \Delta \) is a logical consequence of \( \Gamma \) in an interpretation \( J \) (denoted by \( \Gamma \models J \Delta \)), if \( \bigcap_{\Phi \in \Gamma} T(\Phi J) \cap \bigcap_{\Psi \in \Delta} F(\Psi J) = \emptyset \). \( \Delta \) is a logical consequence of \( \Gamma \) (denoted by \( \Gamma \models \Delta \)), if \( \Gamma \models J \Delta \) in every inter-
pretation $J$. The set of names (variables) that occur in $\Gamma$ is denoted by $nm(\Gamma)$, this notation is extended for a case of several sets or formulas, say, $nm(\Gamma, \Delta, R^x_x(\exists x \Phi))$.

General properties of the consequence relation are the following ($\Gamma, \Delta, \Sigma, \Lambda \subseteq Fr(\Sigma_f^U)$):

C) Let $\Gamma \cap \Delta \neq \emptyset$, then $\Gamma \models \Delta$.
U) Let $\Gamma \subseteq \Lambda$ and $\Delta \subseteq \Sigma$, then $\Gamma \models \Delta \Rightarrow \Lambda \models \Sigma$.

General properties with their duals (co-rules) are the following ($\Gamma, \Delta, \Sigma \subseteq Fr(\Sigma_f^U)$, $\Phi, \Psi \in Fr(\Sigma_f^U)$):

$v_j$) $\Phi \wedge \Psi, \Gamma \models \Delta \Rightarrow \Phi, \Gamma |\models \Delta \Rightarrow \Psi, \Gamma |\models \Delta$;
$v_j$) $\Gamma |\models \Delta \Rightarrow \Phi, \Phi |\models \Psi, \Psi |\models \Delta$;
$RT_j$) $R^\tau_\tau(\Phi, \Gamma) \models \Delta \Leftrightarrow R^\tau_\tau(\Phi), \Gamma |\models \Delta$;
$RT_j$) $\Gamma |\models \Delta, R^\tau_\tau(\Phi) \Leftrightarrow \Gamma |\models \Delta, R^\tau_\tau(\Phi)$;
$\Phi N_j$) $R^\tau_\tau(\Phi, \Gamma) |\models \Delta \Rightarrow R^\tau_\tau(\Phi), \Gamma |\models \Delta$, if $\phi \in U nm(\Phi)$;
$\Phi N_j$) $\Gamma |\models \Delta, R^\tau_\tau(\Phi) \Leftrightarrow \Gamma |\models \Delta, R^\tau_\tau(\Phi)$, if $\phi \in U nm(\Phi)$;
$RR_j$) $R^\tau_\tau(\Phi, \Gamma) |\models \Delta \Leftrightarrow R^\tau_\tau(\Phi), \Gamma |\models \Delta$;
$RR_j$) $\Gamma |\models \Delta, R^\tau_\tau(\Phi) \Leftrightarrow \Gamma |\models \Delta, R^\tau_\tau(\Phi)$;
$R\neg_j$) $R^\tau_\tau(\neg \Phi, \Gamma) |\models \Delta \Leftrightarrow \neg R^\tau_\tau(\Phi), \Gamma |\models \Delta$;
$R\neg_j$) $\Gamma |\models \Delta, R^\tau_\tau(\neg \Phi) \Leftrightarrow \Gamma |\models \Delta, \neg R^\tau_\tau(\Phi)$;
$R\vee_j$) $R^\tau_\tau(\Phi \vee \Psi, \Gamma) |\models \Delta \Leftrightarrow R^\tau_\tau(\Phi \vee \Psi), \Gamma |\models \Delta$;
$R\vee_j$) $\Gamma |\models \Delta, R^\tau_\tau(\Phi \vee \Psi) \Leftrightarrow \Gamma |\models \Delta, R^\tau_\tau(\Phi \vee \Psi)$;
$R\exists \Phi_j$) $R^\tau_\tau(\exists x \Phi, \Gamma) |\models \Delta \Leftrightarrow R^\tau_\tau(\exists x \Phi), \Gamma |\models \Delta$;
$R\exists \Phi_j$) $\Gamma |\models \Delta, R^\tau_\tau(\exists x \Phi) \Leftrightarrow \Gamma |\models \Delta, R^\tau_\tau(\exists x \Phi)$;
$R\exists \Phi_j$) $R^\tau_\tau(\exists x \Phi, \Gamma) |\models \Delta \Leftrightarrow \exists x \Phi, \Gamma |\models \Delta$;
$R\exists \Phi_j$) $\Gamma |\models \Delta, R^\tau_\tau(\exists x \Phi) \Leftrightarrow \Gamma |\models \Delta, \exists x \Phi$.

Properties related to elimination of quantifiers are the following:
$\exists \Phi_j$) $R^\tau_\tau(\exists x \Phi, \Gamma) |\models \Delta \Leftrightarrow R^\tau_\tau(\exists x \Phi, \Gamma) |\models \Delta, \exists x \Phi$, if $\phi \in U \cup nm(\Gamma, \Delta, R^\tau_\tau(\exists x \Phi))$;
$\exists \Phi_j$) $\exists x \Phi, \Gamma |\models \Delta \Leftrightarrow R^\tau_\tau(\exists x \Phi, \Gamma) |\models \Delta, \exists x \Phi$; if $\phi \in U \cup nm(\Gamma, \Delta, \exists x \Phi)$;
$\exists \Phi_j$) $\exists x \Phi |\models \Gamma |\models \Delta, \exists x \Phi, R^\tau_\tau(\exists x \Phi, \Gamma) |\models \Delta$;
$\exists \Phi_j$) $\exists x \Phi, \Gamma |\models \Delta, \exists x \Phi, R^\tau_\tau(\exists x \Phi, \Gamma) |\models \Delta$;
$\exists \Phi_j$) $\exists x \Phi, \Gamma |\models \Delta, \exists x \Phi, R^\tau_\tau(\exists x \Phi, \Gamma) |\models \Delta$;
$\exists \Phi_j$) $\exists x \Phi, \Gamma |\models \Delta, \exists x \Phi, R^\tau_\tau(\exists x \Phi, \Gamma) |\models \Delta$;
$\exists \Phi_j$) $\exists x \Phi, \Gamma |\models \Delta, \exists x \Phi, R^\tau_\tau(\exists x \Phi, \Gamma) |\models \Delta$;
$\exists \Phi_j$) $\exists x \Phi, \Gamma |\models \Delta, \exists x \Phi, R^\tau_\tau(\exists x \Phi, \Gamma) |\models \Delta$.

Let us prove, for example, the property $\exists \Phi_j$ ($\exists \Phi_j$ is its special case) and the property $\exists \Phi_j$ ($\exists \Phi_j$ is its special case) considering an arbitrary interpretation $J$.

Property $\exists \Phi_j$.

$\Rightarrow$. If $R^\tau_\tau(\exists x \Phi, \Gamma) |\models \Delta$ then $T(\Gamma) \cap T(R^\tau_\tau(\exists x \Phi, \Gamma)) \cap F(\Delta) = \emptyset$. Since $T(R^\tau_\tau(\exists x \Phi, \Gamma)) \cap F(\Delta) = \emptyset$, we have $T(\Gamma) \cap F(\Delta) = \emptyset$. Assume that $T(\Gamma) \cap T(R^\tau_\tau(\exists x \Phi, \Gamma)) \cap F(\Delta) = \emptyset$ and there exists $d$ such that $d \in T(\Gamma)$ and $T(R^\tau_\tau(\exists x \Phi, \Gamma)) \cap F(\Delta)$. In this case $d \not\in T(\Gamma)$ and $d \in F(\Delta)$. By $d \not\in T(\Gamma)$ and $d \in F(\Delta)$, we have $d \not\in T(\Gamma)$ and $d \not\in F(\Delta)$; this means that $d \not\in T(\Gamma)$ and $d \not\in F(\Delta)$ for some $d \not\in T(\Gamma)$ and $d \not\in F(\Delta)$. Since $\phi \in U \cap nm(\Gamma, \Delta, R^\tau_\tau(\exists x \Phi))$ we have $d \not\in T(\Gamma)$ and $d \not\in F(\Delta)$; this follows from the property $\exists \Phi_j$ ($\exists \Phi_j$); by definition of $\exists x \Phi$ we have $d \not\in T(\Gamma)$ and $d \not\in F(\Delta)$, therefore $d \not\in T(\Gamma)$ and $d \not\in F(\Delta)$. This contradicts to the assumption that $T(\Gamma) \cap T(R^\tau_\tau(\exists x \Phi, \Gamma)) \cap F(\Delta) = \emptyset$.
\( \Rightarrow \) If \( \Gamma \models \Delta, R^\Sigma_\set stere = (\exists x \Phi) \), then \( \Gamma \models \Delta, R^\Sigma_\set stere = (\exists x \Phi), R^{\Sigma, x}_\set stere = (\Phi), \) \( \) and \( \varepsilon y, \Gamma \models \Delta, R^\Sigma_\set stere = (\exists x \Phi) \) by the general property \( U \).

\( \Leftarrow \) Assume that \( \varepsilon y, \Gamma \models \Delta, R^\Sigma_\set stere = (\exists x \Phi) \) and \( \Gamma \models \Delta, R^\Sigma_\set stere = (\exists x \Phi), R^{\Sigma, x}_\set stere = (\Phi), \varepsilon y, \) but \( \Gamma \models \Delta, R^\Sigma_\set stere = (\exists x \Phi) \). Then we have

\[ T(\Gamma) \cap F(\Delta) \cap F(R^\Sigma_\set stere = (\exists x \Phi)_J) \neq \emptyset; \] this implies that there exists \( d \) such that \( d \in T(\Gamma) \cap F(\Delta) \cap F(R^\Sigma_\set stere = (\exists x \Phi)_J) \).

Two cases are possible: \( (d(y) \uparrow \) and \( d(y) \downarrow \). If \( d(y) \uparrow \) then \( d \in T(e(y)) \); from this \( d \in T(e(y)) \cap T(\Gamma) \cap F(\Delta) \cap F(R^\Sigma_\set stere = (\exists x \Phi)_J) \) that contradicts to \( e(y, \Gamma \models \Delta, R^\Sigma_\set stere = (\exists x \Phi) \). If \( d(y) \downarrow \) then \( d \in F(e(y)) \). Let \( d(y) = a \). By \( d \in F(R^\Sigma_\set stere = (\exists x \Phi)_J) \) we have

\[ d \cap x \rightarrow d(\cap x) \in F(\exists x \Phi)_J. \]

From this follows \( d \cap x \rightarrow d(y) \cap x \rightarrow a \in F(\Phi)_J \), therefore \( d \cap x \rightarrow d(y) \cap x \rightarrow d(y) \in F(\Phi)_J \), thus, \( d \in F(R^\Sigma_\set stere = (\exists x \Phi)_J) \). So, \( d \in T(\Gamma) \cap F(\Delta) \cap F(R^\Sigma_\set stere = (\exists x \Phi)_J) \) and \( F(R^\Sigma_\set stere = (\exists x \Phi)_J) \cap F(e(y)) \) that contradicts \( \Gamma \models \Delta, R^\Sigma_\set stere = (\exists x \Phi), R^{\Sigma, x}_\set stere = (\Phi), \varepsilon y \).

6 The sequent calculus for \( L^\Omega \)

For the logic \( L^\Omega \) we build a calculus of sequent type. Sequents are interpreted as sets of labeled (signed) formulas marked by one of two symbols \(-\) or \(-\). Such sequents \( \Sigma \) are also denoted by \( \Gamma, \Delta \), where all formulas of \( \Gamma \) are labeled by the symbol \(-\) (such formulas are called \( T \)-formulas), of \( \Delta \) – by the symbol \(-\) (\( F \)-formulas). This notation for sequents is similar to notations used in tableau calculi.

Semantic properties of relation \( \models \) have their syntactic analogues – sequent rules. These rules are the following.

Sequent rules for propositional compositions:

\[ \vdash R^\Sigma_\set stere = (\Phi, \Sigma); \quad \vdash R^\Sigma_\set stere = (\Phi, \Sigma); \]

\[ \vdash R^\Sigma_\set stere = (\Phi, \Sigma); \quad \vdash R^\Sigma_\set stere = (\Phi, \Sigma); \]

\[ \vdash R^\Sigma_\set stere = (\Phi \vee \Psi, \Sigma); \quad \vdash R^\Sigma_\set stere = (\Phi \vee \Psi, \Sigma); \]

Sequent rules for renomination compositions:

\[ \vdash R^\Sigma_\set stere = (\exists x \Phi, \Sigma); \quad \vdash R^\Sigma_\set stere = (\exists x \Phi, \Sigma); \]

\[ \vdash R^\Sigma_\set stere = (\exists x \Phi, \Sigma); \quad \vdash R^\Sigma_\set stere = (\exists x \Phi, \Sigma); \]

\[ \vdash \neg R^\Sigma_\set stere = (\Phi, \Sigma); \quad \vdash \neg R^\Sigma_\set stere = (\Phi, \Sigma); \]

\[ \vdash \neg R^\Sigma_\set stere = (\Phi, \Sigma); \quad \vdash \neg R^\Sigma_\set stere = (\Phi, \Sigma); \]

Sequent rules for quantification compositions:

\[ \vdash R^\Sigma_\set stere = (\Phi), \Sigma; \quad \vdash R^\Sigma_\set stere = (\Phi), \Sigma; \]

\[ \vdash R^\Sigma_\set stere = (\Phi), \Sigma; \quad \vdash R^\Sigma_\set stere = (\Phi), \Sigma; \]

\[ \vdash R^\Sigma_\set stere = (\Phi \vee \Psi), \Sigma; \quad \vdash R^\Sigma_\set stere = (\Phi \vee \Psi), \Sigma; \]

\[ \vdash R^\Sigma_\set stere = (\exists x \Phi), \Sigma; \quad \vdash R^\Sigma_\set stere = (\exists x \Phi), \Sigma; \]

\[ \vdash R^\Sigma_\set stere = (\exists x \Phi), \Sigma; \quad \vdash R^\Sigma_\set stere = (\exists x \Phi), \Sigma; \]

\[ \vdash R^\Sigma_\set stere = (\exists x \Phi), \Sigma; \quad \vdash R^\Sigma_\set stere = (\exists x \Phi), \Sigma; \]
algebra-based logics constructed in a semantic-syntactic style on the methodological basis that is common with program-ness. Program-oriented logics developed in the paper are called composition-nominative program logics and are soundness and completeness.

In our previous work we constructed a calculus for different neoclassical logics and proved their soundness and completeness [8]. Similar results were also proved for some classes of composition-nominative modal and temporal logics [8]. The obtained results can be used in logics for program reasoning.

7 Conclusions

In the paper we have advocated the idea that logics for program reasoning should be based directly on formal program models. In this case program logics should reflect such program features as partiality, complex data structures, nondeterminism etc. Program-oriented logics developed in the paper are called composition-nominative program logics and are algebra-based logics constructed in a semantic-syntactic style on the methodological basis that is common with program-
ming; they can be considered as generalizations of traditional logics on classes of partial predicates that do not have fixed arity. Such predicates, called quasiary predicates, are defined over partial variable assignments (partial data). We have described the hierarchy of different logics of quasiary predicates. For the constructed logics some laws of classical logic fail because of partiality of predicates and data. We have constructed a sequent calculus for a first-order composition-nominative pure quasiary predicate logic which plays the central role in the logic hierarchy. We have proved soundness and completeness of this calculus. The obtained results can be generalized for a number of more powerful logics. The proposed methods can be useful for construction and investigation of logics for program reasoning.

Future work on the topic will include construction of sequent calculi for composition-nominative logics over hierarchic nominative data. Hierarchic data permit to represent such complex structures as lists, stacks, arrays, etc.; thus, such logics will be closer to program models with more rich data types. Also, prototypes of software systems for theorem proving in composition-nominative logics should be developed.

References