LOGICS OF QUASIARY PREDICATES IN FORMAL SOFTWARE DEVELOPMENT

Mykola S. Nikitchenko and Valentyn G. Tymofieiev

Department of Theory and Technology of Programming
Taras Shevchenko National University of Kyiv
64, Volodymyrska Street, 01601 Kyiv, Ukraine
nikitchenko@unicyb.kiev.ua tvalentyn@univ.kiev.ua

Abstract

Logics of quasiary predicates are algebra-based logics constructed in a semantic-syntactic style on the methodological basis, which is common with programming. Such way of construction gives a possibility to use these logics in formal methods of software development. We demonstrate the main ideas of program logic construction and give formal definitions for an important fragment of program logics called many-sorted first-order composition-nominative logics. These logics are generalizations of classical logics for a case of partial predicates that do not have fixed arity; reasoning rules for such logics differ from classical ones. We study semantic properties of these logics used for investigation of satisfiability and validity problems.

Keywords - Software development, formal methods, partial predicate, partial logic, composition-nominative logic, first-order logic, satisfiability, validity.

1 INTRODUCTION

Formal methods of software (program) development aim to increase software reliability by using a number of mathematically based approaches that permit to prove correctness of software (and hardware) system components with respect to a formal specification. Application of such methods should be based on formal program models and on a logic for reasoning about programs. This explains the necessity of construction and investigation of such logics called *program logics.* Many such logics are used in computer science for the specification of software and hardware systems; for example: Hoare logic, Hennessy-Milner logic, dynamic logic, modal and temporal logics, separation logic, spatial logic, nominal logic, linear logic, etc. [1]. It is not possible to invent one universal program logic that would have all necessary properties. Therefore a hierarchy of logics has to be developed. Here we continue our work on construction of such a hierarchy. Constructed logics are based directly on special program models called *composition-nominative program models* (CNPM) [2]. Corresponding logics are called *composition-nominative program logics* (CNPL).

Program models and logics are constructed according to the principles of *development from abstract to concrete*, *priority of semantics*, *compositionality*, and *nominativity*.

These principles specify a hierarchy of new logics that are semantically based on algebras of predicates, functions, and programs, which are considered as partial mappings. Operations over such mappings are called *compositions*. Data classes are considered on various abstraction levels, but the main attention is paid to the class of *nominative data*. Such data consist of pairs *name–value*. Nominative data can represent various data structures such as records, arrays, lists, relations, etc. [2]; this explains the importance of the notion of nominative data. In the simplest case nominative data can be treated as partial mappings from a certain set *V* of (possibly typed) names (or variables) into a set of basic (atomic) values. Such data are called *nominative sets.* Nominative sets represent program states for simple programming languages (see, for example, [3]). From this follows that semantic models of programs and logics are mathematically based on the notion of nominative set (nominative data in general case). Partial mappings over nominative sets are called *quasiary*. This fact permits to integrate models of programs and logics, and represent them as a hierarchy of composition-nominative models [4]. Logics developed within such approach are called *composition-nominative program logics* because such logics are determined 1) by operations (*compositions*) in algebras of partial predicates, functions, and programs, and 2) by *nominative* structures of data on which these predicates, functions and programs are defined. Let us admit that such properties as partiality and compositionality are recognized as important for computer science [5, 6].

In this paper we introduce the notion of many-sorted CNPL and give formal definitions for its fragment called many-sorted first-order composition-nominative logic. This logic is an extension of a many-sorted composition-nominative pure predicate logic (a logic without functions) studied in [7]. We formulate semantic properties of such logics used for investigation of satisfiability and validity problems.

The paper is structured in the following way. In section 2 we give a motivating example; then in section 3 we give formal definitions of the many-sorted logics under investigation, and define the satisfiability and validity problems. In section 4 we study semantic properties of such logics. In section 5 we summarize our results and indicate directions for future investigations.

2 CONSTRUCTING PROGRAM LOGICS: A MOTIVATING EXAMPLE

Let us consider a simple programming language ELT (Example Language with Types) which is used here to demonstrate how many-sorted program logics can be constructed. The ELT version presented here is an extension of ELT from [7].

2.1 A simple programming language ELT

The grammar of the language is defined as follows:

prg ::= begin var *dcl* ; *stm* end

dcl ::= *i*: integer | *x*: real | *dcl* ; *dcl*

stm ::= *i*:=*ie* | *x*:=*re* | stm_1 ; stm_2 | if *b* then stm_1 else stm_2 |
while *b* do *stm* | begin *stm* end

ie ::= *k* | *i* | $ie_1 + ie_2$ | $ie_1 - ie_2$ | $ie_1 * ie_2$ | ie_1 *div* ie_2 | ie_1 *mod* ie_2 | (*ie*)

re ::= *c* | *x* | $re_1 + re_2$ | $re_1 - re_2$ | $re_1 * re_2$ | re_1 / re_2 | (*re*)

b ::= $ie_1 = ie_2$ | $ie_1 > ie_2$ | $re_1 = re_2$ | $re_1 > re_2$ | $b_1 \vee b_2$ | $\neg b$ | (*b*)

where:

- *k* ranges over integer numbers *Int*={..., –2, –1, 0, 1, 2, …};
- *c* ranges over real numbers *Real*={ …, –0.1, …, 0.0, …, 0.1, …};
- *i* ranges over variables (names) of integer type V_I={*I*, *M*, *N*, …};
- *x* ranges over variables (names) of real type V_R={*R*, *X*, *Y*, …};
- *ie* ranges over integer expressions *Iexpr*;
- *re* ranges over real expressions *Rexpr*;
- *b* ranges over Boolean expressions *Bexpr*;
- *stm* ranges over statements *Stm*;
- *dcl* ranges over variable declarations *Dcl*;
- *prg* ranges over programs *Prg*.

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

```
begin
        var N: integer; X: real; R: real;
                R:=1.0;
                while N>0 do
                  if (N mod 2)=1
                    then begin R:=R*X; N:=N–1 end
                    else begin X:=X*X; N:=N div 2 end
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 *many-sorted program algebra with n-ary mappings* oriented on the program *ES*;
- 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 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 (N–1, N *div* 2, N *mod* 2, 1.0, $R*X$, $X*X$) expressions, and 3) statements obtained with the help of structuring constructs such as assignment, conditional, and loop. Note that *div* is partial on *Int*. To emphasize mapping's partiality/totality we write the sign $\xrightarrow{p}$ for partial mappings and the sign $\xrightarrow{t}$ for total mappings.

We use denotational semantics (see, e.g. [3]) 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 *biquasiary* mappings.

2.2 Classes of *n*-ary mappings

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

For our language ELT we define the following types of *n*-ary mappings:

$$Fn^{n,Int}=Int^{n} \xrightarrow{p} Int,\ Pr^{n,Int}=Int^{n} \xrightarrow{p} Bool,\ Pr^{n,Bool}= Bool^{n} \xrightarrow{p} Bool,$$

$$Fn^{n,Real}=Real^{n} \xrightarrow{p} Real,\ Pr^{n,Real}= Real^{n} \xrightarrow{p} Bool,\ n\geq 0.$$

Using the same notation for language symbols of various types and mappings they represent, we can write that +, –, $*$, *div*, *mod*: $Fn^{2,Int}$; +, –, $*$, /: $Fn^{2,Real}$; =, >: $Pr^{2,Int}$; =, >: $Pr^{2,Real}$; $\vee$: $Pr^{2,Bool}$, $\neg$: $Pr^{1,Bool}$.

2.3 Classes of quasiary mappings

Quasiary mappings are defined over classes of states considered as sets of named values. For example, the expression $R*X$ specifies a function which given a state d of the form [$R \mapsto r$, $X\mapsto x$], where r and x are real numbers, evaluates a value $r*x$. Examples of states are [$X\mapsto$ 8.3, $N \mapsto$ 4], [$X\mapsto$ 8.3, $N \mapsto$ 4, $R \mapsto$ 8.2], [$X\mapsto$ 8.3]. In a state d a variable v can have a value (this is denoted $d(v)\downarrow$) or be undefined (denoted $d(v)\uparrow$); thus, [$X\mapsto$ 8.3, $N \mapsto$ 4](X)$\downarrow$ and [$X\mapsto$ 8.3, $N \mapsto$ 4](R)$\uparrow$.

Formally, states are defined in the following way. Let V={N, X, R} be the set of variables, T ={*Int*, *Real*} be the set of types, A=*Int*$\cup$*Real* be the set of all values. *Type valuation* (*type assignment*) mapping τ_{ES} : $V \xrightarrow{t} \mathrm{T}$ is τ_{ES} =[$N\mapsto$ *Int*, $X\mapsto$ *Real*, $R\mapsto$ *Real*]. Now, having τ_{ES} we can define the set of states $State_{ES}$ as the set of all partial mappings d:$V \xrightarrow{p} A$ such that the value of N in d belongs to *Int* if it is defined and the values of X and R belong to *Real* if they are defined.

Having described states we are able to represent formal semantics of Boolean and arithmetic expressions. Boolean expressions denote predicates (called *many-sorted quasiary predicates*) of the set $Pr(\tau_{ES})=State_{ES} \xrightarrow{p} Bool$; thus for $b\in Bexpr$ we have $[\![b]\!] \in Pr(\tau_{ES})$. Integer expressions denote functions (called *many-sorted quasiary functions of integer type*) of the set $Fn^{Int}(\tau_{ES})=State_{ES} \xrightarrow{p} Int$; thus, for $ie\in Iexpr$ we have $[\![ie]\!] \in Fn^{Int}(\tau_{ES})$. Real expressions denote functions (called *many-sorted quasiary functions of real type*) of the set $Fn^{Real}(\tau_{ES})=State_{ES} \xrightarrow{p} Real$; thus, for $re\in Rexpr$ we have $[\![re]\!] \in Fn^{Real}(\tau_{ES})$. As states are constructed with the help of naming (nominative) relation, they are also called *typed nominative sets* and their class is denoted by $NST(\tau_{ES})$. Functions from $Fn^{Int}(\tau_{ES})$ and $Fn^{Real}(\tau_{ES})$ 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 denomination (denaming) functions $'x$: $Fn^{Real}(\tau_{ES})$ for variables of real type and $'i$: $Fn^{Int}(\tau_{ES})$ for variables of integer type. For a given program state, the function $'x$ (or $'i$) returns the value of the variable x (or i) in that state. For instance, denomination function that yields the value of name N is denoted by $'N$. Such values may or may not be defined, so denomination functions are partial.

Semantics of numbers is treated as quasiary constant functions of corresponding types. Such functions are represented by constants $\mathbf{c}$ and $\mathbf{k}$ written in bold font; thus, $[\![1.0]\!] = \mathbf{1.0}$, $[\![1]\!] = \mathbf{1}$. It is clear that $\mathbf{1.0} \in Fn^{Real}(\tau_{ES})$ and $\mathbf{1} \in Fn^{Int}(\tau_{ES})$.

For specifying semantics of complex expressions special compositions called *superpositions* are used. A superposition $S^n_{Real} : Fn^{n,Real} \times (Fn^{Real}(\tau_{ES}))^n \xrightarrow{t} Fn^{Real}(\tau_{ES})$ of quasiary functions $g_1,\ldots, g_n$ into an n-ary function $f^n_{Real} \in Fn^{n,Real}$ is an operator such that

$$S^n_{Real}\ (f^n_{Real}\ , g_1,\ldots, g_n)(d) = f^n_{Real}\ (g_1(d),\ldots,g_n(d))$$

where d is a state. The same formulas can be used for superpositions of other types, for which we adopt the following notations: S^n_{Int} is a superposition into an n-ary function of integer type, $S^n_{P,Int}$ ($S^n_{P,Real}$) is a superposition into n-ary predicate over Int (over $Real$), $S^n_{P,Bool}$ is a superposition into n-ary Boolean function. Thus,

$$[\![N\text{–}1]\!] = S^2_{Int}(–, 'N, \mathbf{1}),\quad [\![N>0]\!] = S^2_{P,Int}(>, 'N, \mathbf{0}),\quad [\![(N\ mod\ 2)=1]\!] = S^2_{P,Int}(=, S^2_{Int}\ (mod, 'N, \mathbf{2}), \mathbf{1}).$$

So, semantics of ELT expressions can be presented via introduced mappings and superposition compositions.

2.4 Classes of biquasiary (program) functions

The semantics of statements (programs) is represented by *biquasiary* functions (also called *program functions*). Their class is denoted $FPrg(\tau_{ES}) = State_{ES} \xrightarrow{p} State_{ES} = NST(\tau_{ES}) \xrightarrow{p} NST(\tau_{ES})$. Program functions are constructed with the help of special compositions. The following compositions with conventional meaning are used for formalizing semantics of ELT:

1. integer assignment composition $AS^i_I: Fn^{Int}(\tau_{ES}) \xrightarrow{t} FPrg(\tau_{ES})$ and real assignment composition $AS^x_R : Fn^{Real}(\tau_{ES}) \xrightarrow{t} FPrg(\tau_{ES})$ (parameter i denotes a variable of integer type and parameter x denotes a variable of real type);
2. composition of sequential execution $\bullet$: $FPrg(\tau_{ES}) \times FPrg(\tau_{ES}) \xrightarrow{t} FPrg(\tau_{ES})$;
3. conditional composition IF: $Pr(\tau_{ES}) \times FPrg(\tau_{ES}) \times FPrg(\tau_{ES}) \xrightarrow{t} FPrg(\tau_{ES})$;
4. loop composition WH: $Pr(\tau_{ES}) \times FPrg(\tau_{ES}) \xrightarrow{t} FPrg(\tau_{ES})$.

Note, that we define $\bullet$ by commuting arguments of conventional functional composition: $f \bullet g = g \circ f$. Thus, $[\![R{:=}R{*}X;\ N{:=}N\text{-}1]\!] = AS^R_R\ (S^2_{Real}\ (*, 'R, 'X)) \bullet AS^N_I\ (S^2_{Int}\ (–, 'N, \mathbf{1}))$.

2.5 Program algebra with *n*-ary mappings

The definitions introduced permit to conclude that the following *many-sorted program algebra with n-ary mappings* oriented on ES type assignment mapping has been constructed (we omit types of compositions):

$$\boldsymbol{A}^n(\tau_{ES}) = < Fn^{2,Int}, Pr^{2,Int}, Pr^{2,Bool}, Pr^{1,Bool}, Fn^{2,Real}, Pr^{2,Real}, Pr(\tau_{ES}), Fn^{Int}(\tau_{ES}), Fn^{Real}(\tau_{ES}),$$
$$FPrg(\tau_{ES}); +, –, *, div, mod: Fn^{2,Int}; +, –, *, /: Fn^{2,Real}; =, >: Pr^{2,Int}; =, >: Pr^{2,Real}; \vee: Pr^{2,Bool}, \neg : Pr^{1,Bool},$$
$$\mathbf{c}, \mathbf{k}, 'x, 'i, S^2_{Int}, S^2_{Real}, S^2_{P,Int}, S^2_{P,Real}, S^2_{P,Bool}, S^1_{P,Bool}, AS^i_I, AS^x_R, \bullet, IF, WH> .$$

Note that notation for parametric compositions (like denominations, assignments etc.) represents here classes of compositions. Thus, $'x$ represents the class of compositions for various x.

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

The term for *ES* is as follows:

$$WH(S^2_{P,Int}(>, 'N, \mathbf{0}), IF(S^2_{P,Int}\ (=, S^2_{Int}\ (mod, 'N, \mathbf{2}), \mathbf{1}),$$

$$AS^R_R\ (S^2_{Real}\ (*, 'R, 'X)) \bullet AS^N_I\ (S^2_{Int}\ (-, 'N, \mathbf{1})),$$

$$AS^X_R\ (S^2_{Real}\ (*, 'X, 'X)) \bullet AS^N_I\ (S^2_{Int}\ (div, 'N, \mathbf{2})))).$$

Note that this term and its sub-terms can denote partial mappings as the 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 $R:=R*X$ and $N:=N-1$ by proving in the algebra $\boldsymbol{A}^n(\tau_{ES})$ the corresponding property of composition of sequential execution:

$$AS^R_R\ (S^2_{Real}\ (*, 'R, 'X)) \bullet AS^N_I\ (S^2_{Int}\ (-, 'N, \mathbf{1})) = AS^N_I\ (S^2_{Int}\ (-, 'N, \mathbf{1}))\ \bullet AS^R_R\ (S^2_{Real}\ (*, 'R, 'X))\ .$$

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.

2.6 Program algebra of quasiary 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 the 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 $\boldsymbol{x*y}$ (printed in bold font) such that $\boldsymbol{x*y} = S^2_{Real}\ (*, 'x, 'y)$; here *x* and *y* are parameters from *V*.

Therefore instead of binary functions we introduce parametric quasiary functions ***x+y***, ***x–y***, ***x*y***, and ***x/y*** over *Real* and ***n+m***, ***n–m***, ***n*m***, ***n div m***, and ***n mod m*** over *Int*; also instead of relations we introduce new parametric quasiary predicates ***x=y***, ***x>y***, ***n=m***, and ***n>m*** (with *x*, *y*, *n*, and *m* as parameters).

This step permits to represent every *n*-ary function defined over *Real* or *Int* as a parametric quasiary function. But now, to represent the semantics of complex expressions we should introduce special superpositions $S^{v_1,\ldots v_n}_{Real}$, $S^{v_1,\ldots v_n}_{Int}$ (or $S^{\bar{v}}_{Real}$, $S^{\bar{v}}_{Int}$) and $S^{v_1,\ldots v_n}$ (or $S^{\bar{v}}$), which are called *superpositions into real quasiary function*, *integer quasiary function*, and quasiary *predicate* respectively. Superposition composition is represented by a formula $S^{v_1,\ldots v_n}_{Real}(f^q, g_1,\ldots, g_n)(d) = f^q(d\nabla[v_1 \mapsto g_1(d),\ldots,v_n \mapsto g_n(d)])$ where $f^q \in Fn^{Real}(\tau_{ES})$ and ∇ denotes state updating operation [4]. Intuitive meaning of this formula is that we change in *d* the values of names $v_1,\ldots, v_n$ to $g_1(d),\ldots,g_n(d)$ respectively (partiality should be taken into account). Thus, semantics, say of the expression $R+(R*X)$, can be represented as $S^Y_{Real}\ (\boldsymbol{R+Y}, \boldsymbol{R*X})$.

Now let us consider logical symbols $\vee$ and $\neg$ treated earlier as Boolean functions of the types $Bool^2 \xrightarrow{t} Bool$ and $Bool \xrightarrow{t} Bool$. We cannot directly represent them as quasiary predicates defined on program states, therefore we advocate another approach. We will treat them as the following binary compositions over quasiary predicates (denoted by the same signs):

$$\vee: Pr(\tau_{ES}) \times Pr(\tau_{ES}) \xrightarrow{t} Pr(\tau_{ES}) \text{ and } \neg: Pr(\tau_{ES}) \xrightarrow{t} Pr(\tau_{ES}).$$

Such representations also provide better possibilities to work with partial predicates. For example, consider a Boolean expression $(M>N) \vee (M>L)$. Its semantics in $\boldsymbol{A}^n(\tau_{ES})$ is represented by the term $S^2_{P,Bool}\ (\vee, S^2_{P,Int}(>, 'M, 'N), S^2_{P,Int}\ (>, 'M, 'L))$. But superposition into an *n*-ary mapping is strict: when one argument is not defined then the result is also undefined. This restricts usage of such superpositions. For example, for Kleene's strong disjunction it is allowed that one argument may be undefined if the other one is evaluated to true. Therefore superpositions cannot help in formalizing strong connectives, but when representing connectives as compositions, we avoid this difficulty.

Thus, we can now consider a simpler algebra – the program algebra of quasiary predicates with algebra constants:

$$\boldsymbol{A}^{q}(\tau_{ES}) = < Pr(\tau_{ES}),\ Fn^{Int}(\tau_{ES}),\ Fn^{Real}(\tau_{ES}),\ FPrg(\tau_{ES});\ \boldsymbol{n{+}m},\ \boldsymbol{n{-}m},\ \boldsymbol{n{*}m},\ \boldsymbol{n\ div\ m},\ \boldsymbol{n\ mod\ m},\ \boldsymbol{x{+}y},$$
$$\boldsymbol{x{-}y},\ \boldsymbol{x{*}y},\ \boldsymbol{x/y},\ \boldsymbol{n{=}m},\ \boldsymbol{n{>}m},\ \boldsymbol{x{=}y},\ \boldsymbol{x{>}y},\ \boldsymbol{c},\ \boldsymbol{k},\ \vee,\ \neg,\ 'x,\ 'i,\ S^{\bar{v}}_{Int},\ S^{\bar{v}}_{Real},\ S^{\bar{v}},\ AS^{i}_{I},\ AS^{x}_{R},\ \bullet,\ IF,\ WH > .$$

This algebra is simpler than $\boldsymbol{A}^{n}(\tau_{ES})$, but it is still associated with specifics of *ES* program and ELT language. To make the algebra more general we should distinguish between descriptive and logical symbols of our language.

Trivial inspection of definitions shows that algebra constants ***n+m***, ***n–m***, ***n*m***, ***n div m***, ***n mod m***, ***x+y***, ***x–y***, ***x*y***, ***x/y***, ***n=m***, ***n>m***, ***x=y***, ***x>y***, ***c***, ***k*** are descriptive symbols because they represent specifics of real and integer numbers. Other symbols may be considered logical.

Thus, we can make the next step of constructing more "logical" algebras by eliminating descriptive symbols having predefined interpretations. We obtain a new program algebra:

$$\boldsymbol{A}(\tau_{ES}) = < Pr(\tau_{ES}),\ Fn^{Int}(\tau_{ES}),\ Fn^{Real}(\tau_{ES}),\ FPrg(\tau_{ES});$$
$$\vee,\ \neg,\ 'x,\ 'i,\ S^{\bar{v}}_{Int},\ S^{\bar{v}}_{Real},\ S^{\bar{v}},\ AS^{i}_{I},\ AS^{x}_{R},\ \bullet,\ IF,\ WH > .$$

As to descriptive symbols, we can instead consider sets *Fs* and *Ps* of function and predicate symbols that do not have predefined interpretations, and consequently, can denote any quasiary function or predicate.

Being interested in general laws of reasoning about programs, we should make the next step and define compositions for any type valuation mapping $\tau : V \xrightarrow{t} \mathrm{T}$ where *V* is a set of names and T is a class of types. In this case we obtain the following program algebra of quasiary predicates:

$$\boldsymbol{A}(\tau) = < Pr(\tau),\ \{Fn^{A}(\tau) \mid A \in \mathrm{T}\},\ FPrg(\tau);\ \vee,\ \neg,\ 'x,\ S^{\bar{v}}_{A},\ S^{\bar{v}},\ AS^{i}_{A},\ \bullet,\ IF,\ WH > .$$

Formal definitions will be given in the next section; parameter *A* belongs to T. Symbols from *Fs* and *Ps* 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 program algebras (for various τ), 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 $FPrg(\tau)$. Such logics define equivalent transformations of programs.

Another conventional program logic is Floyd–Hoare logic, which is based on assertions of the form $\{b_1\}stm\{b_2\}$. Semantics of such assertions can be presented by Floyd–Hoare composition

$$FH{:}Pr(\tau) \times FPrg(\tau) \times Pr(\tau) \xrightarrow{t} Pr(\tau).$$

We define this composition under assumption that predicates and functions can be partial. Then

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

Here we write $p(d)\downarrow$ if a predicate *p* is defined on data *d*, $p(d)\downarrow = r$ if a predicate *p* is defined on data *d* with a value *r*, $p(d)\uparrow$ if a predicate *p* is undefined on *d*.

Extending the algebra $\boldsymbol{A}(\tau)$ with *FH* composition, a parametric composition of existential quantification $\exists x{:}\ Pr(\tau) \xrightarrow{t} Pr(\tau)$, and a compositions of equality $=_A$ we obtain the algebra

$$\boldsymbol{A_{FH}}(\tau) = < Pr(\tau)\, ,\, \{Fn^A(\tau) \mid A \in \mathrm{T}\},\, FPrg(\tau);\, \vee,\, \neg,\, 'x,\, S_A^{\bar v}\, ,\, S^{\bar v}\, ,\, \exists x\, ,\, =_A,\, AS_A^i\, ,\, \bullet,\, IF,\, WH,\, FH > .$$

The class of such algebras forms the semantic base for quite powerful Floyd–Hoare-like logics of quasiary mappings. Investigation of such logics is a special challenge, here we restrict ourselves by studying a fragment of these logics called *many-sorted first-order composition-nominative logic* (L_{FO}). Semantic base for such logics are algebras $\boldsymbol{A_{FO}}(\tau)$ obtained by restricting of $\boldsymbol{A_{FH}}(\tau)$ on the predicate and function carriers:

$$\boldsymbol{A_{FO}}(\tau) = < Pr(\tau)\, ,\, \{Fn^A(\tau) \mid A \in \mathrm{T}\};\, \vee,\, \neg,\, 'x,\, S_A^{\bar v}\, ,\, S^{\bar v}\, ,\, \exists x\, ,\, =_A > .$$

Such algebras play an important role in studying Floyd–Hoare-like logics because many correctness conditions are usually verified for these algebras.

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

3 FORMAL DEFINITIONS OF MANY-SORTED FIRST-ORDER COMPOSITION-NOMINATIVE LOGIC

To define the logic we have to specify its semantic, syntactic, and interpretational components. Semantic component is formed by predicate algebras formally defined below.

3.1 Semantic component of L_{FO}

Let V be a *set of names*. According to tradition, names from V are also called *variables*. Let T be a *class of types* and $\tau : V \xrightarrow{t} \mathrm{T}$ be a total mapping called *type valuation*.

Given V, T, and τ, a class $NST(V, \mathrm{T}, \tau)$ (shortly: $NST(\tau)$) of typed nominative sets is defined by the following formula:

$$NST(\tau) = \left\{ d : V \xrightarrow{p} \bigcup_{A \in \mathrm{T}} A \;\middle|\; \forall v \in V\ (d(v)\downarrow \Rightarrow d(v) \in \tau(v)) \right\}.$$

Informally speaking, typed nominative sets represent states of typed variables.

Let $Pr(V, \mathrm{T}, \tau) = NST(\tau) \xrightarrow{p} Bool$ be the set of all partial predicates (this set is shortly denoted by $Pr(\tau)$). Predicates from $Pr(\tau)$ are called *many-sorted partial quasiary predicates*. Let $A \in \mathrm{T}$. The class of *many-sorted partial quasiary functions into A* is denoted by $Fn^A(V, \mathrm{T}, \tau) = NST(\tau) \xrightarrow{p} A$ (shortly $Fn^A(\tau)$).

Compositions of the algebra under investigation

$$\boldsymbol{A_{FO}}(\tau) = < Pr(\tau)\, ,\, \{Fn^A(\tau) \mid A \in \mathrm{T}\};\, \vee,\, \neg,\, 'x,\, S_A^{\bar v}\, ,\, S^{\bar v}\, ,\, \exists x\, ,\, =_A >$$

are defined in the following way (p, $q \in Pr(\tau)$, $d \in NST(\tau)$).

Disjunction and negation compositions are defined as follows:

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

Existential quantifications are defined as follows ($x \in V$ is a parameter).

$$(\exists x\ p)(d) = \begin{cases} T, \text{ if there exists } a \in \tau(x) : p(d\nabla[x \mapsto a]) \downarrow= T, \\ F, \text{if for each } a \in \tau(x) : p(d\nabla[x \mapsto a]) \downarrow= F, \\ \text{undefined in other cases.} \end{cases}$$

Superpositions $S_A^{v_1,\dots v_n}$, $S^{v_1,\dots v_n}$ with $\tau(v_1) = A_1,\dots,\tau(v_n) = A_n, n \geq 0$ are compositions of types

$$Fn^A(\tau)\times Fn^{A_1}(\tau)\times\dots\times Fn^{A_n}(\tau)\xrightarrow{t}Fn^A(\tau),\ Pr(\tau)\times Fn^{A_1}(\tau)\times\dots\times Fn^{A_n}(\tau)\xrightarrow{t}Pr(\tau)$$

and are evaluated as follows ($f \in Fn^A(\tau)$, $p \in Pr(\tau)$, $g_1 \in Fn^{A_1}(\tau)$, …, $g_n \in Fn^{A_n}(\tau)$):

$$S_A^{v_1,\dots v_n}(f,g_1,\dots,g_n)(d) = f([v \mapsto a \in_n d \mid v \notin \{v_1,\dots,v_n\}]\nabla[v_1 \mapsto g_1(d),\dots,v_n \mapsto g_n(d)]),$$

$$S^{v_1,\dots v_n}(p,g_1,\dots,g_n)(d) = p([v \mapsto a \in_n d \mid v \notin \{v_1,\dots,v_n\}]\nabla[v_1 \mapsto g_1(d),\dots,v_n \mapsto g_n(d)]).$$

Here by $v \mapsto a \in_n d$ *we* denote that the value of v in d is defined and is equal to a.

The denomination composition $'x$ ($x \in V$) has a type $Fn^A(\tau)$ such that $A = \tau(x)$. Given a nominative set d we have that $'x(d) = d(x)$.

The equality composition $=_A$ ($A \in \mathrm{T}$) has a type $Fn^A(\tau)\times Fn^A(\tau) \to Pr(\tau)$ and is defined as follows ($f,g \in Fn^A(\tau)$):

$$=_A (f,g)(d) = \begin{cases} T, \text{ if } f(d)\downarrow,\ g(d)\downarrow, \text{and } f(d) = g(d), \\ T, \text{ if } f(d)\uparrow \text{ and } g(d)\uparrow, \\ F \text{ otherwise.} \end{cases}$$

Now we are able to specify precisely the class of compositions for the algebra $\boldsymbol{A_{FO}}(\tau)$ as a set

$$C_{FO}(\tau) = \{\vee, \neg\} \cup \{S_A^{\bar v} \mid A \in \mathrm{T},\ \bar v = (v_1,\dots,v_n) \text{ is a list of distinct names, } n \geq 0\} \cup$$

$$\{S^{\bar v} \mid \bar v = (v_1,\dots,v_n) \text{ is a list of distinct names, } n \geq 0\} \cup \{\ 'x \mid x \in V\} \cup \{\exists x \mid x \in V\} \cup \{=_A \mid A \in \mathrm{T}\}.$$

Thus, we have defined algebras of the form $\boldsymbol{A_{FO}}(V,\mathrm{T},\tau) =< Pr(\tau), \{Fn^A(\tau) \mid A \in \mathrm{T}\}; C_{FO}(\tau) >$ called *many-sorted first-order algebras of quasiary predicates and functions (semantic algebras)*. For such algebras we use simpler notations $\boldsymbol{A_{FO}}(\tau)$ or $\boldsymbol{A_{FO}}$ if parameters V, T, τ are clear from the context.

3.2 Syntactic component of L_{FO}

Syntactic component of a logic specifies its language.

Let V be a set of names. Let S be a set of sorts, $\xi_V : V \xrightarrow{t} S$ be *a sort valuation mapping*, $\Sigma^S = (V, S, \xi_V)$ be *a signature of sort valuation* such that for all $s \in S$ the set $\{v \in V \mid \xi_V(v) = s\}$ is infinite.

Let us define a set of composition symbols

$$Cs_{FO}(\Sigma^S) = \{\vee, \neg\} \cup \{S_s^{\bar v} \mid s \in S,\ \bar v = (v_1,\dots,v_n) \text{ is a list of distinct names, } n \geq 0\} \cup$$

$$\{S^{\bar v} \mid \bar v = (v_1,\dots,v_n) \text{ is a list of distinct names, } n \geq 0\} \cup \{\ 'x \mid x \in V\} \cup \{\exists x \mid x \in V\} \cup \{=_s \mid s \in S\}.$$

Languages of many-sorted first-order composition-nominative logics (L_{FO}*-languages*) are defined by their signatures Σ_{FO} of the form $\Sigma_{FO} = (\Sigma^S, Cs_{FO}(\Sigma^S), Ps, Fs, \xi_F)$, where Ps is a set of predicate symbols, Fs is a set of function symbols, and $\xi_F : Fs \xrightarrow{t} S$ is a mapping that for every function symbol yields its sort. Given a signature Σ_{FO} we can inductively define the *set of terms* $Tr(\Sigma_{FO})$ and the *set of formulas* $Fr(\Sigma_{FO})$. With every term *t* from the set $Tr(\Sigma_{FO})$ we associate some sort from the set S, which we denote $\xi_T(t)$.

The set $Tr(\Sigma_{FO})$ is defined as follows:

1. If $F \in Fs$, then $F \in Tr(\Sigma_{FO})$, $\xi_T(F) = \xi_F(F)$. Such terms are called *atomic*.
2. If $x \in V$, then $'x \in Tr(\Sigma_{FO})$, $\xi_T('x) = \xi_V(x)$.
3. If $\bar{v} = (v_1,...,v_n)$ is a list of distinct variables, t, $t_1,...,t_n \in Tr(\Sigma_{FO})$, $\xi_T(t) = s, \xi_T(t_1) = \xi_V(v_1)$, …, $\xi_T(t_n) = \xi_V(v_n)$, $n \geq 0$, then $S_s^{\bar{v}}(t, t_1,...,t_n) \in Tr(\Sigma_{FO})$, $\xi_T(S_s^{\bar{v}}(t, t_1,...,t_n)) = s$.

The set $Fr(\Sigma_{FO})$ is defined as follows:

1. If $P \in Ps$ then $P \in Fr(\Sigma_{FO})$. Such formulas are called *atomic*.
2. If $\Phi, \Psi \in Fr(\Sigma_{FO})$ then $(\Phi \vee \Psi) \in Fr(\Sigma_{FO})$ and $\neg\Phi \in Fr(\Sigma_{FO})$.
3. If $\bar{v} = (v_1,...,v_n)$ is a list of distinct variables, $\Phi \in Fr(\Sigma_{FO})$, $t_1,...,t_n \in Tr(\Sigma_{FO})$, $\xi_T(t_1) = \xi_V(v_1)$, …, $\xi_T(t_n) = \xi_V(v_n)$, $n \geq 0$, then $S^{\bar{v}}(\Phi, t_1,...,t_n) \in Fr(\Sigma_{FO})$.
4. If $\Phi \in Fr(\Sigma_{FO})$, $x \in V$, then $\exists x\, \Phi \in Fr(\Sigma_{FO})$.
5. If $t_1, t_2 \in Tr(\Sigma_{FO})$, $\xi_T(t_1) = \xi_T(t_2) = s$, then $=_s (t_1, t_2) \in Fr(\Sigma_{FO})$. Taking into account that a sort for a term can be identified unambiguously, we will write $=_s (t_1, t_2)$ in a shorter form $t_1 = t_2$.

An L_{FO}-language of a signature $\Sigma_{FO} = (\Sigma^S, Cs_{FO}(\Sigma^S), Ps, Fs, \xi_F)$ is defined by a set of formulas $Fr(\Sigma_{FO})$. Such formulas are also called L_{FO}-formulas.

3.3 Interpretational component of L_{FO}

Let $\Sigma_{FO} = (\Sigma^S, Cs_{FO}(\Sigma^S), Ps, Fs, \xi_F)$ be a signature of an L_{FO}-language, $\Sigma^S = (V, S, \xi_V)$, $\boldsymbol{A_{FO}}(V, \mathrm{T}, \tau) = < Pr(\tau), \{Fn^A(\tau) \mid A \in \mathrm{T}\}; C_{FO}(\tau) >$ be an arbitrary semantic algebra, $I^S : S \xrightarrow{t} \mathrm{T}$ be a *sort interpretation mapping* such that $\tau = I^S \circ \xi_V$. Let $I^{Ps} : Ps \xrightarrow{t} Pr(\tau)$ be a *predicate symbols interpretation mapping*, $I^{Fs} : Fs \xrightarrow{t} \bigcup_{A \in \mathrm{T}} Fn^A(\tau)$ be a mapping called *function symbols interpretation mapping* such that for every $f \in Fs$ $I^{Fs}(f)) \in Fn^A(\tau)$ if $I^S(\xi_F(f)) = A$. A tuple *J*=($\boldsymbol{A_{FO}}$, Σ_{FO}, I^S, I^{Ps}, I^{Fs}) is called *an interpretation of the* L_{FO}-language of the signature Σ_{FO}.

There is a correspondence between composition symbols from $Cs_{FO}(\Sigma^S)$ and compositions from $C_{FO}(\tau)$ of the algebra $\boldsymbol{A_{FO}}$. For simplicity's sake we use the same notations for composition symbols and for compositions that correspond to them. Therefore, we will consider composition symbols as interpreted.

Given an L_{FO}-formula $\Phi \in Fr(\Sigma_{FO})$, a term $t \in Tr(\Sigma_{FO})$, and a Σ_{FO}-interpretation *J*, an interpretation mechanism maps t to some quasiary function $t_J \in Fn(\tau)$ and Φ to some quasiary predicate $\Phi_J \in Pr(\tau)$. Values of atomic terms and formulas are defined by interpretational mappings I^{Fs}, I^{Ps}. Meanings of more complicated constructions are defined inductively according to definitions of composition operations. Compositions symbols that have a sort *s* as a parameter correspond to the composition with a parameter $I^S(s)$. For example, with every symbol of equality composition $=_s \in Cs_{FO}(\Sigma^S)$ we associate an equality composition $=_A \in C_{FO}(\tau)$, where $A = I^S(s)$.

Interpretational component of the logic L_{FO} is defined by the class of all possible interpretations of L_{FO}-language.

An L_{FO}-formula $\Phi \in Fr(\Sigma_{FO})$ is said to be *satisfiable on a* Σ_{FO}*-interpretation J* if there is an element d from the domain of Φ_J such that $\Phi_J(d)\downarrow = T$. This is denoted as $J \mid\approx \Phi$.

An L_{FO}-formula $\Phi \in Fr(\Sigma_{FO})$ is said to be *satisfiable in* L_{FO} if there is a Σ_{FO}-interpretation J such that $J \mid\approx \Phi$. This is denoted $\mid\approx_{FO} \Phi$.

An L_{FO}-formula $\Phi \in Fr(\Sigma_{FO})$ is said to be *valid on a* Σ_{FO}*-interpretation J* if there is no element d from the domain of Φ_J such that $\Phi_J(d)\downarrow = F$. This is denoted $J \models \Phi$.

An L_{FO}-formula $\Phi \in Fr(\Sigma_{FO})$ is said to be *valid in* L_{FO} if $J \models \Phi$ for every Σ_{FO}-interpretation J. This is denoted $\models_{FO} \Phi$.

Satisfiability problem for L_{FO} consists in checking whether $\mid\approx_{FO} \Phi$ holds for arbitrary L_{FO}-formula Φ. Validity problem for L_{FO} consists in checking whether $\models_{FO} \Phi$ holds for arbitrary L_{FO}-formula Φ.

Two formulas $\Phi, \Psi \in Fr(\Sigma_{FO})$ are equivalent (denoted $\Phi \approx \Psi$) if $\Phi_J = \Psi_J$ for every Σ_{FO}-interpretation J. Two terms $t_1, t_2 \in Tr(\Sigma_{FO})$ are equivalent (denoted $t_1 \approx t_2$) if $(t_1)_J = (t_2)_J$ for every Σ_{FO}-interpretation J.

4 SEMANTIC PROPERTIES OF L_{FO}

Formal definitions of L_{FO} give a possibility to study properties of L_{FO}-formulas and their equivalent transformations. Such properties can be used to develop a sequent calculus for L_{FO} in style of [4] and prove its soundness and completeness or to transform L_{FO}-formulas to equisatisfiable formulas of classical many-sorted first-order logic in style of [7]. Therefore, existent state-of-the-art methods and techniques for checking satisfiability and validity in classical logics can also be applied to composition-nominative logics.

It should be noted that L_{FO} may be treated as a generalization of classical logic on a case of partial predicates that do not have fixed arity. But for this logic *some reasoning rules of classical logic fail*, for example *modus ponens* is not valid because of partiality of predicates, the law $(\forall x\Phi) \to \Phi$ is not valid because not all variables have values (partiality of data), etc. It means that such logics require special investigations.

Here we only formulate some semantic properties of L_{FO}: interrelation of validity and satisfiability, monotonicity of compositions, logic extension with unessential variables, equivalent formula transformations.

Due to possible presence of a nowhere defined predicate (which is a valid predicate) we do not have in L_{FO} the property that a formula Φ is satisfiable if Φ is valid (which holds for classical first-order logic). But reduction of satisfiability to validity still holds in L_{FO}: for any Σ_{FO}-interpretation J formula Φ is satisfiable in J iff $\neg\Phi$ is not valid in J.

Monotonicity of compositions means that being applied to extended predicates (or functions) compositions yield extended results. This property is useful for such transformation of partial predicates to total predicates that preserve validity or satisfiability of corresponding formulas [7]. Monotonicity is also important for studying recursion in first-order predicate algebras.

Unessential variables that do not affect the meanings of predicates or functions are important for equivalent transformations of formulas. Such variables play a role of “additional memory”; their class for L_{FO} is denoted by U [8].

The main equivalent transformations for L_{FO} are formulated as follows ($\bar{v}, \bar{x}, \bar{u}$ are lists of distinct variables, $\bar{t}, \bar{r}, \bar{q}, \bar{w}$ are lists of terms, other symbols are understood in a usual way, also we assume that formulas are built correctly with respect to sorts of their components and parameters):

E1. $S^{\bar{v}}(\Phi\vee\Psi,\bar{t}) \approx S^{\bar{v}}(\Phi,\bar{t}) \vee S^{\bar{v}}(\Psi,\bar{t})$.

E2. $S^{\bar{v}}(\neg\Phi,\bar{t}) \approx \neg S^{\bar{v}}(\Phi,\bar{t})$.

E3. $S^{\bar{v}}(\exists x\Phi,\ \bar{t}) \approx \exists u\ S^{\bar{v}}(S^{x}(\Phi,'u),\ \bar{t})$, u is unessential variable that does not occur in $S^{\bar{v}}(\exists x\Phi,\bar{t})$, $u \in U$, $\xi_V(u) = \xi_V(x)$.

E4. $S^{\bar{u},\bar{x}}(S^{\bar{x},\bar{v}}(\Phi,\ \bar{r},\bar{q}),\ \bar{t},\bar{w}) \approx S^{\bar{u},\bar{x},\bar{v}}(\Phi,\ \bar{t},\ S^{\bar{u},\bar{x}}_{s_1}(r_1,\ \bar{t},\ \bar{w}), \ldots,\ S^{\bar{u},\bar{x}}_{s_k}(r_k,\ \bar{t},\ \bar{w}),\ S^{\bar{u},\bar{x}}_{s'_1}(q_1,\ \bar{t},\ \bar{w}), \ldots,$ $S^{\bar{u},\bar{x}}_{s'_m}(q_m,\ \bar{t},\ \bar{w}))$, here and in E5 $\bar{u} = u_1, \ldots, u_n$; $\bar{t} = t_1, \ldots, t_n$; $\bar{x} = x_1, \ldots, x_k$; $\bar{r} = r_1, \ldots, r_k$; $\bar{w} = w_1, \ldots, w_k$; $\bar{v} = v_1, \ldots, v_m$; $\bar{q} = q_1, \ldots, q_m$, $u_i \neq v_j, i = 1,\ldots,n, j = 1,\ldots,m$; the sorts in superpositions are defined by the sorts of substituted terms.

E5. $S^{\bar{u},\bar{x}}_{s}(S^{\bar{x},\bar{v}}_{s}(t,\ \bar{r},\bar{q}),\ \bar{t},\bar{w}) \approx S^{\bar{u},\bar{x},\bar{v}}_{s}(t,\ \bar{t},\ S^{\bar{u},\bar{x}}_{s_1}(r_1,\ \bar{t},\ \bar{w}), \ldots,\ S^{\bar{u},\bar{x}}_{s_k}(r_k,\ \bar{t},\ \bar{w}),\ S^{\bar{u},\bar{x}}_{s'_1}(q_1,\ \bar{t},\ \bar{w}), \ldots,$ $S^{\bar{u},\bar{x}}_{s'_m}(q_m,\ \bar{t},\ \bar{w}))$.

E6. $S^{\bar{v}}(r = q,\ \bar{t}) \approx S^{\bar{v}}_{s}(r,\ \bar{t}) = S^{\bar{v}}_{s}(q,\ \bar{t})$.

E7. $S^{\bar{v}}(\Phi,\ \bar{t}) \approx S^{x,\bar{v}}(\Phi,'x,\bar{t})$, x does not occur in $\bar{v}$. In particular, $\Phi \approx S^{x}(\Phi,\ 'x)$.

E8. $S^{\bar{v}}_{s}(t,\ \bar{t}) \approx S^{x,\bar{v}}_{s}(t,\ 'x,\ \bar{t})$, x does not occur in $\bar{v}$. In particular, $t \approx S^{x}_{s}(t,\ 'x)$.

E9. $S^{\bar{u},x,\bar{v}}(\Phi,\bar{q},r,\bar{w}) \approx S^{x,\bar{u},\bar{v}}(\Phi,\ r,\bar{q},\bar{w})$.

E10. $S^{\bar{u},x,\bar{v}}_{s}(t,\bar{q},r,\bar{w}) \approx S^{x,\bar{u},\bar{v}}_{s}(t,\ r,\bar{q},\bar{w})$.

E11. $S^{x,\bar{v}}_{s}('x,t,\bar{r}) \approx t$.

E12. $S^{\bar{v}}_{s}('x,\bar{r}) \approx 'x$, x does not occur in $\bar{v}$.

The equivalences E1-E12 describe transformations of formulas which permit to obtain equivalent formulas in pseudoclassical form; generalizing techniques developed in [7, 8] further transformations of such formulas can be proposed that permit to obtain formulas of first-order classical logic which preserve the main properties of initial formulas. Thus, many standard methods developed for classical logics can be applied for investigation of composition-nominative logics. The authors plan to present such techniques in forthcoming papers.

5 CONCLUSIONS

Formal approaches for software systems development can be based on different logics. In this paper we have discussed the composition-nominative approach, its motivation, and program logics evolved in this approach. Such logics are constructed in a semantic-syntactic style on the methodological basis, which is common with programming. They are algebra-based logics of partial predicates, ordinary and program functions that do not have fixed arity. We have given formal definitions for an important fragment of program logics called many-sorted first-order composition-nominative logics. These logics are generalizations of classical logics but their reasoning rules differ from classical ones. We have formulated semantic properties of these logics, which are useful for investigation of satisfiability and validity problems.

Future work on the topic will include construction of sequent calculus for many-sorted first-order composition-nominative logic and investigation of properties of this calculus. Another direction of investigation concerns special composition-nominative program logics, in particular, logics of Floyd-Hoare style with simple and structured data types.

References

[1] Handbook of Logic in Computer Science, S. Abramsky, Dov M. Gabbay, and T. S. E. Maibaum (eds.), in 5 volumes, Oxford Univ. Press, Oxford (1993–2001)

[2] Nikitchenko, N.S.: A Composition Nominative Approach to Program Semantics. Technical Report IT–TR 1998-020, Technical University of Denmark (1998)

[3] Winskel G.: The Formal Semantics of Programming Languages. MIT Press (1993)

[4] Nikitchenko M.S., Shkilniak S.S.: Mathematical logic and theory of algorithms. Publishing house of Taras Shevchenko National University of Kyiv, Kyiv (in Ukrainian) (2008)

[5] Blamey, S.: Partial Logic. In Gabbay D., Guenthner F. (eds.), Handbook of Philosophical Logic, Volume III, D. Reidel Publishing Company (1986)

[6] Janssen T.M.V.: Compositionality. In van Benthem J., ter Meulen A. (eds.), Handbook of Logic and Language, pp. 417-473. Elsevier and MIT Press (1997)

[7] Nikitchenko M., Tymofieiev V.: Satisfiability and Validity Problems in Many-Sorted Composition-Nominative Pure Predicate Logics. In: V. Ermolayev et al. (eds.): ICTERI 2012, CCIS 347, pp. 89-110. Springer, Heidelberg (2012)

[8] Nikitchenko M.S., Tymofieiev V.G.: Satisfiability in Composition-Nominative Logics. Central European Journal of Computer Science, vol. 2, issue 3, pp. 194-213 (2012)