Developing Composition-Nominative Program Logics

Mykola (Nikolaj) S. Nikitchenko

Taras Shevchenko National University of Kyiv

Contents

- Introduction
- Program logic construction
- First order composition-nominative logic
- Reduction of the satisfiability problem
- Conclusions

Formal methods in system development

They are used in case of safety-critical components.

Degrees of formality:

- formal specification only;
- formal specification and rigorous development, the refinement relations are not proved formally;
- formal specifications and formal development, the refinement relations are formally proved.

Application of formal methods requires program logics.

Aim:

to construct logic more adequate to programming problems

Discrepancies between traditional logic and programming

- semantics of programs - partial functions whereas in traditional logic - total functions and predicates;
- programming languages have a developed system of data types whereas traditional logic - simple unstructured types (sorts);
- semantic aspects of programs prevail over syntactical aspects whereas in traditional logic we have an inverse situation.
- non-determinism: programs can evaluate to different results on the same input data

Thus, semantics-based (algebra-based) approach is proposed for logic construction.

Principles of the approach

- **Development** from abstract to concrete;
- **Priority of semantics** over syntax
- **Compositionality**: properties of complex systems (programs) are specified by properties of their components;
- **Nominativity** (from Latin *nomen* – name): emphasizes the importance of naming relations in program specification and construction.

Development chain

As a result - **a hierarchy of composition-nominative program algebras** (CNPA).

Then - **composition-nominative program logics** are constructed directly based on CNPA

At last - **compositional methods of program verification.**

In shorter form:

program algebras $\Rightarrow$

program logics $\Rightarrow$

program verification

Example Language EL: BNF

<program>::= begin <command> end

<command> ::= <var>:= <a_expr> | <command> ; <command> |
if <b_expr> then <command> else <command> |
while <b_expr> do <command> | begin <command> end

<a_expr> ::= <number> | <var> | <a_expr> + <a_expr> |
<a_expr> –<a_expr> | <a_expr> * <a_expr> | (<a_expr>)

<b_expr> ::=<a_expr>= <a_expr> |<a_expr> > <a_expr> |
<b_expr> ∨ <b_expr> |¬ <b_expr> | (<b_expr>)

<var>::= M | N | ...

<number>::= ... -2 | -1 | 0 | 1 | 2 | ...

EL syntax: "mathematical" notation

prg::= begin c end
c::= x:=a | $c1$; $c2$| if b then $c1$ else $c2$ |while b do c |
begin c end | skip
a::= n | x | $a1 + a2$ | $a1 - a2$ | $a1 * a2$ | (a)
b::=$a1= a2$ |$a1>a2$ | $b1 \vee b2$ |$\neg b$ | (b),

where:

- n ranges over natural numbers $Nat=\{0, 1, 2, \ldots\}$,
- x ranges over variables (names) $V=\{M, N, \ldots\}$,
- a ranges over arithmetic expressions *Aexpr*,
- b ranges over Boolean expressions *Bexpr*,
- c ranges over commands (statements) (programs) *Cmn*,
- *prg* ranges over programs *Prg*.

GCD

```
begin
   while ¬(M=N) do
            if M>N then
                  M:=M–N
            else
                  N:=N–M
end
```

How can we represent program semantics?

n-ary functions

$$Fn^{n,Nat} = Nat^{n} \xrightarrow{p} Nat,$$

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

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

$+, -, * : Fn^{2,Nat}$,
$=, > : Pr^{2,Nat}$,
$\vee : Pr^{2,Bool}$, $\neg : Pr^{1,Bool}$.

States and Quasiary Mappings

$State = V \xrightarrow{p} Nat$ (denoted ${}^{V}Nat$).

Quasiary mappings (nominative mappings) over V and Nat

$Fn^{V,Nat} = State \xrightarrow{p} Nat = {}^{V}Nat \xrightarrow{p} Nat$

- for semantics of arithmetical expressions

$Pr^{V,Nat} = {}^{V}Nat \xrightarrow{p} Bool$;

- for semantics of Boolean expressions

Superpositions

$$S_F^n : Fn^{n,Nat} \times (Fn^{V,Nat})^n \xrightarrow{t} Fn^{V,Nat}$$

$$S_P^n : Pr^{n,Nat} \times (Fn^{V,Nat})^n \xrightarrow{t} Pr^{V,Nat}$$

$$S_B^n : Pr^{n,Bool} \times (Pr^{V,Nat})^n \xrightarrow{t} Pr^{V,Nat}$$

$$S_F^n (f^n, g_1, \ldots, g_n)(d) = f^n (g_1(d), \ldots, g_n(d))$$, etc.

Semantics of expressions

Denaming function $'x$: $Fn^{V,Nat}$.

$'x$ returns the value of the variable x in the given state

The denomination function is also called denaming function and is denoted $x\Rightarrow$.

$$[\![N–M]\!] = S^2_F(–, 'M, 'N),$$

$$[\![N>M]\!] = S^2_P(>, 'M, 'N),$$

$$[\![(N>M)\vee(M>N)]\!] = S^2_B(\vee,\ S^2_P(>, 'N, 'M),\ S^2_P(>, 'N, 'M)).$$

Compositions for EL constructs:

1. assignment AS^x: $Fn^{V,Nat} \xrightarrow{t} Prg^{V,Nat}$
2. sequential execution $\bullet$: $Prg^{V,Nat} \times Prg^{V,Nat} \xrightarrow{t} Prg^{V,Nat}$;
3. condition: IF: $Pr^{V,Nat} \times Prg^{V,Nat} \times Prg^{V,Nat} \xrightarrow{t} Prg^{V,Nat}$;
4. loop WH: $Pr^{V,Nat} \times Prg^{V,Nat} \xrightarrow{t} Prg^{V,Nat}$.

Example:

$[\![M:=M–N]\!] = AS^M(S_F^2 (–, 'M, 'N)),$

$[\![M:=M–N; N:=M–N]\!] = AS^M(S_F^2 (–, 'M, 'N)) \bullet AS^N(S_F^2 (–, 'M, 'N)).$

Definition of compositions

Composition	Formula
Superposition	$(S_T^n(f, g_1, \ldots, g_n))(st)=f(g_1(st), \ldots, g_n(st)),\ T \in \{B, P, F\}$
Assignment	$AS^x(fa)(st)=st\ \nabla[x \mapsto fa(st)]$
Sequential execution	$fs_1 \bullet fs_2(st)=fs_2(fs_1(st))$
Conditional operator	$IF(fb, fs1, fs2)(st) = \begin{cases} fs1(st), & \text{if} \quad fb(st) = true, \\ fs2(st), & \text{if} \quad fb(st) = false. \end{cases}$
Loop	$WH(fb,fs)(st)=st_n$, where $st_0=st$, $st_1=fs(st_0)$, $st_2=fs(st_1)$, $\ldots$, $st_n=fs(st_{n-1})$, such that $fb(st_0)=true$, $fb(st_1)=true, \ldots,$ $fb(st_{n-1})=true$, $fb(st_n)=false$.
Denomination function	$x{\Rightarrow}(st)=st(x)$ (or $'x\ (st)=st(x)$)
Identity function	$id(st)=st$

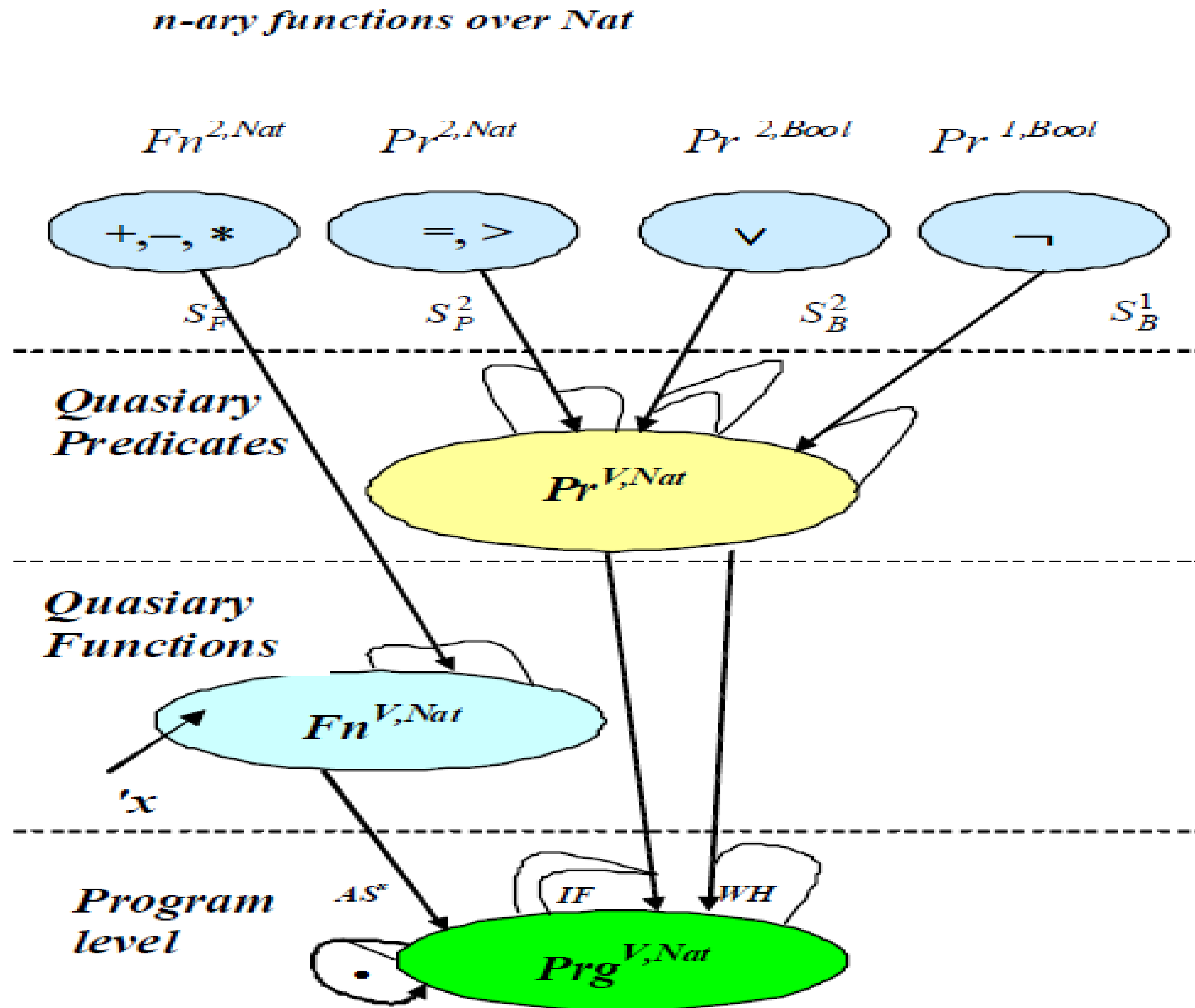
Program algebra with *n*-ary functions

We obtain program algebra with *n*-ary functions:

$$APn(V) = < Fn^{2,Nat}, Pr^{2,Bool}, Pr^{1,Bool}, Pr^{2,Nat},$$
$$Fn^{V,Nat}, Pr^{V,Nat}, Prg^{V,Nat};$$
$$+, -, *, \vee, \neg, =, >,$$
$$S_F^2, S_P^2, S_B^2, S_B^1,$$
$$'x, AS^x, \bullet, IF, WH > .$$

Semantics of EL programs can be represented as terms of this algebra.

Program algebra with *n*-ary functions



Construction of semantic terms

$[\![\text{begin } c \text{ end}]\!] = [\![c]\!]$

$[\![x{:=}a]\!] = AS^x(\,[\![a]\!]\,)$

$[\![c_1\,;\,c_2]\!] = [\![\,c_1]\!] \bullet [\![c_2\,]\!]$

$[\![\text{ if } b \text{ then } c_1 \text{ else } c_2\,]\!] =$

$= IF(\,[\![b]\!]\,,\,[\![c_1]\!]\,,\,[\![c_2\,]\!]\,)$

$[\![\text{while } b \text{ do } c\,]\!] = WH(\,[\![b]\!]\,,\,[\![c]\!]\,)$

$[\![\text{begin } c \text{ end}]\!] = [\![c]\!]$

$[\![skip]\!] = id$

$[\![n]\!] = \overline{n}$

$[\![\,x\,]\!] = {}'x \;\; (\text{or } x\Rightarrow)$

$[\![a_1 + a_2]\!] = S_F^2\,(+,\,[\![a_1]\!]\,,\,[\![a_2]\!]\,)$

$[\![a_1 - a_2]\!] = S_F^2\,(-,\,[\![a_1]\!]\,,\,[\![a_2]\!]\,)$

$[\![a_1 * a_2]\!] = S_F^2\,(*,\,[\![a_1]\!]\,,\,[\![a_2]\!]\,)$

$[\![(a)]\!] = [\![a]\!]$

$[\![b_1 = b_2]\!] = S_P^2\,(=,\,[\![b_1]\!]\,,\,[\![b_2]\!]\,)$

$[\![b_1 > b_2]\!] = S_P^2\,(>,\,[\![b_1]\!]\,,\,[\![b_2]\!]\,)$

$[\![b_1 \vee b_2]\!] = S_B^2\,(\vee,\,[\![b_1]\!]\,,\,[\![b_2]\!]\,)$

$[\![\neg b]\!] = S_B^1\,(\neg,\,[\![\,b]\!]\,)$

GCD semantics

The term for EL program GCD:

$$WH(S_B^1(\neg, S_P^2(=, 'M, 'N)),$$
$$IF(\ S_P^2(>, 'M, 'N),$$
$$AS^M(S_F^2(-, 'M, 'N)),$$
$$AS^M(S_F^2(-, 'N, 'M)))).$$

GCD

while $\neg(M=N)$ do
  if $M>N$ then
    $M:=M-N$
  else
    $N:=N-M$

Proof of program properties

Associativity of sequential execution of statements:

`begin` s_1 ; s_2 `end`; s_3 = s_1; `begin` s_2 ; s_3 `end`

In the algebra $APn(V)$: $f\bullet(g\bullet h)=(f\bullet g)\bullet h$.

Distributivity in condition:

`begin if` b `then` s_1 `else` s_2 `end`; s_3 =

= `if` b `then begin` s_1; s_3 `end else begin` s_2; s_3 `end`.

Corresponding property of $APn(V)$:

$IF(p,f,g)\bullet h=IF(p, f\bullet h, g\bullet h)$. (Here $f, g, h \in Fn^{V,Nat}$, $p\in Pr^{V,Nat}$.)

The second phase of program algebra development

- to make the algebra simpler
- exclude the classes of *n*-ary functions and predicates,
- concentrate on logical symbols that are interpreted as compositions over nominative carriers

$$Fn^{V,Nat}, Pr^{V,Nat}, Prg^{V,Nat}.$$

Program algebra over nominative mappings (with constants)

$Fn^{V,Nat} = State \xrightarrow{p} Nat = {}^{V}Nat \xrightarrow{p} Nat;$

$Pr^{V,Nat} = State \xrightarrow{p} Bool = {}^{V}Nat \xrightarrow{p} Bool;$

Parametric quasiary functions and predicates:

$x-y,\ x+y,\ x*y\ ;\ x=y,\ x>y\ ;$

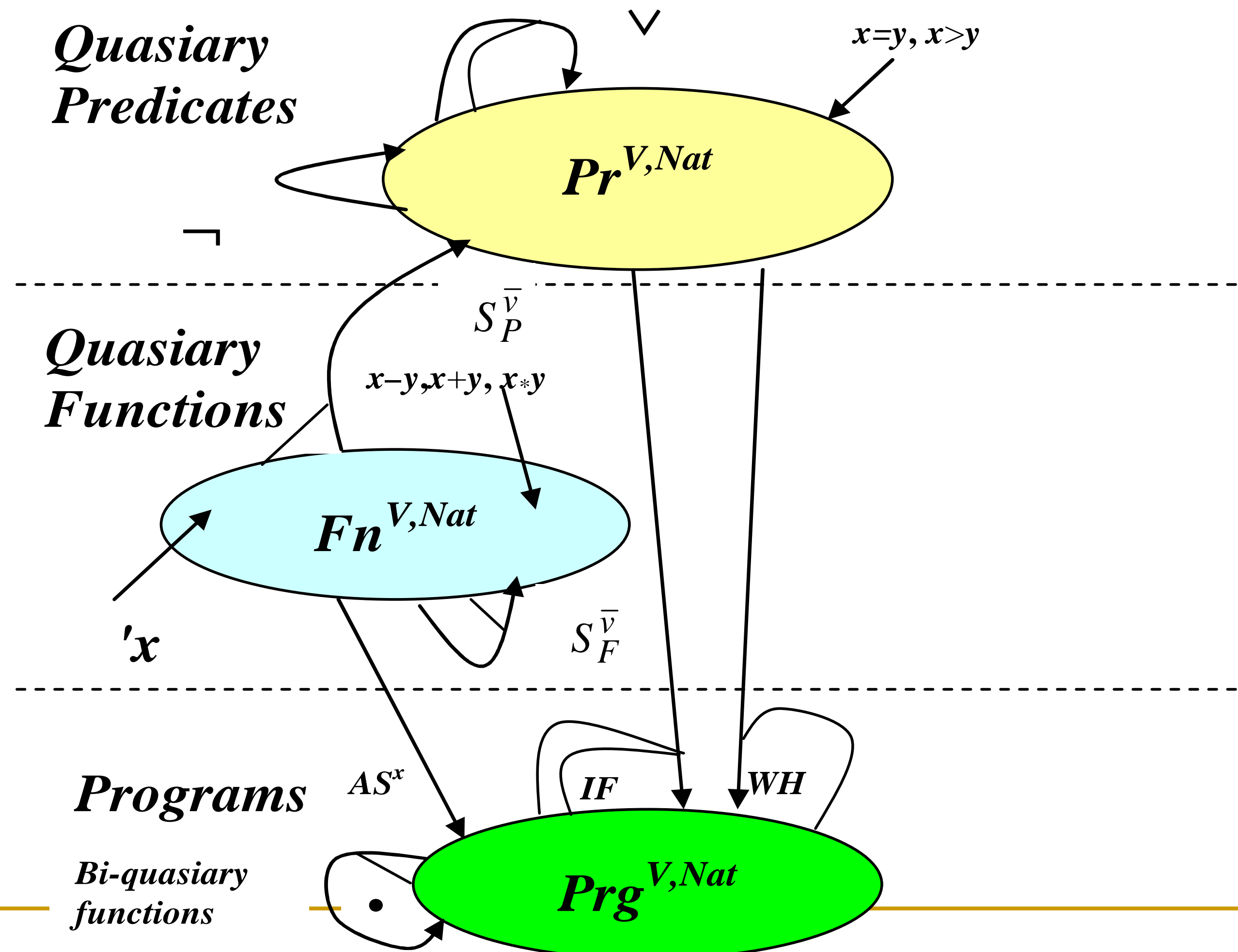
Algebra:

$APQC(V) = < Fn^{V,Nat}, Pr^{V,Nat}, Prg^{V,Nat};$

$x-y,\ x+y,\ x*y\ ;\ x=y,\ x>y\ ;$

$\vee,\ \neg,\ S_F^{\bar{v}},\ S_P^{\bar{v}},\ 'x,\ AS^x,\ \bullet,\ IF,\ WH>.$

Program algebra over nominative mapping (with constants)



GCD

while $\neg(M=N)$ do

  if $M>N$ then

    $M:=M-N$

  else

    $N:=N-M$

Term in APQC:

$$WH(\ \neg\,(\boldsymbol{M}=\boldsymbol{N}),\ IF(\boldsymbol{M}>\boldsymbol{N},\ AS^{M}(\boldsymbol{M}-\boldsymbol{N})),\ AS^{M}(\boldsymbol{N}-\boldsymbol{M}))).$$

cf. Term in APn

$$WH(S_B^1(\neg, S_P^2(=, 'M, 'N)),\ IF(\ S_P^2(>, 'M, 'N),\ AS^M(S_F^2(-, 'M, 'N)),\ AS^M(S_F^2(-, 'N, 'M)))).$$

Program algebra of quasiary mappings (without constants)

Next step: eliminate descriptive symbols with fixed interpretations ($x–y, x+y, x*y, x=y, x>y$)
Consider set of function symbols Fs and predicate symbols Ps

We obtain a program algebra of quasiary mappings $APQ(V)$:

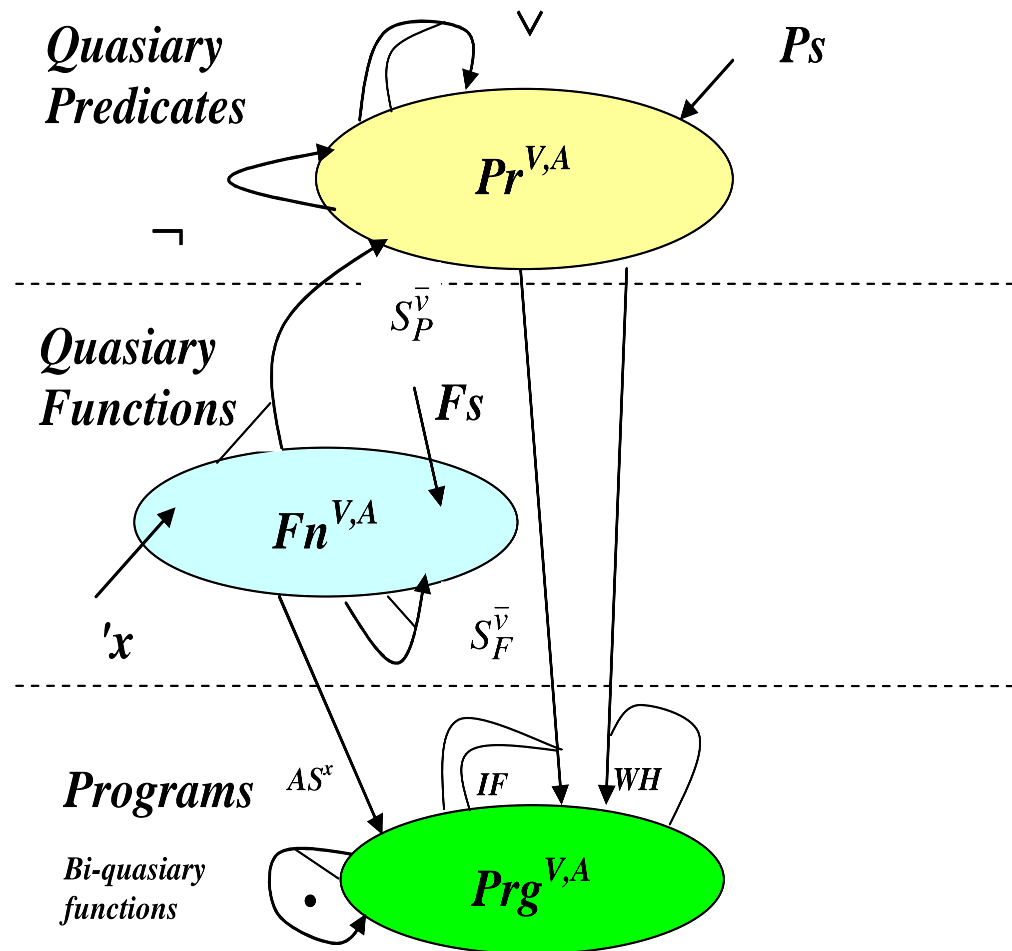
$$APQ(V) = < Fn^{V,Nat}, Pr^{V,Nat}, Prg^{V,Nat};$$
$$\vee, \neg, S_F^{\bar{v}}, S_P^{\bar{v}}, 'x, AS^x, \bullet, IF, WH >.$$

Class of Program Algebras

Next step: *Nat* is change on any *A*

$$APQ(V, A) = < Fn^{V,A}, Pr^{V,A}, Prg^{V,A};$$
$$\vee, \neg, S_F^{\bar{v}}, S_P^{\bar{v}}, {}'x, AS^x, \bullet, IF, WH >.$$

Class of Program Algebras



From Program Algebras to Program Logics

- Add compositions that relates programs with predicates, e.g. Floyd-Hoare composition

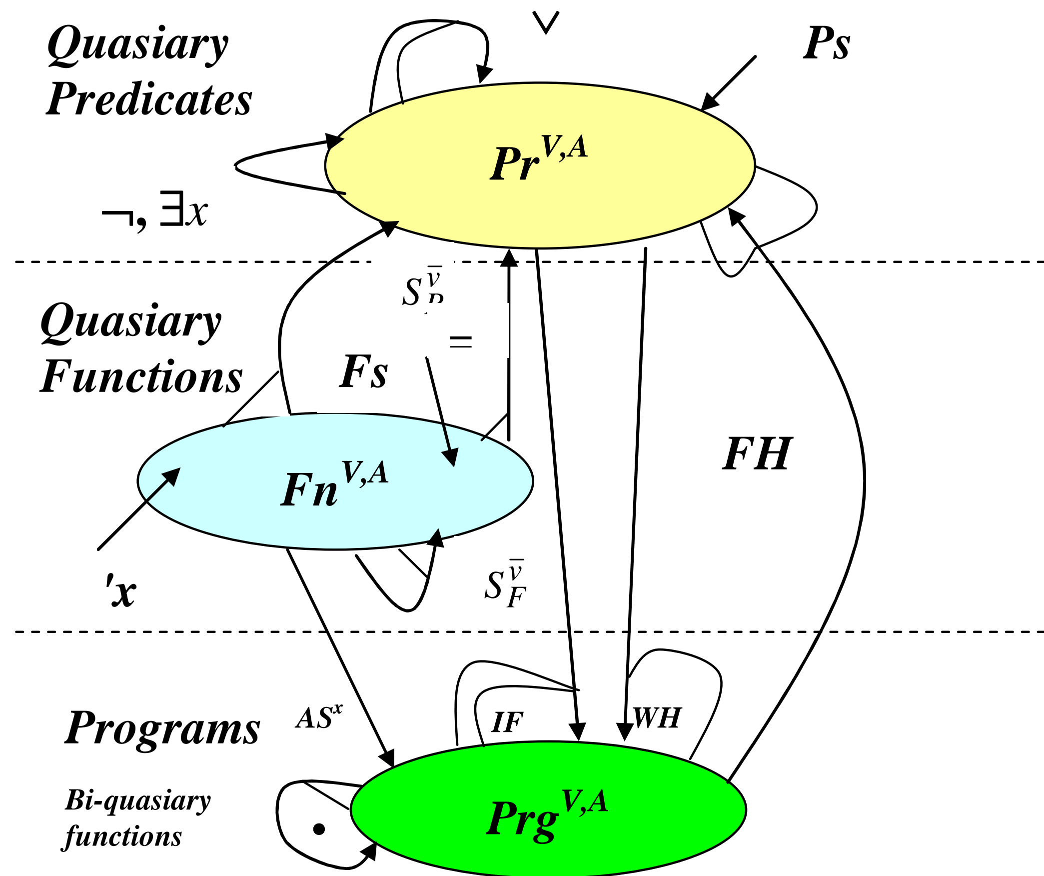
Floyd–Hoare composition FH: $Pr^{V,A} \times Prg^{V,A} \times Pr^{V,A} \xrightarrow{t} Pr^{V,A}$
for partial functions and predicates:

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

- Add quantifiers
- Add equality

Then program logic is obtained (semantic aspects).

Composition-Nominative Program Algebra (Logics) of Floyd-Hoare type



First-order Composition-Nominative Algebras

- First-order algebra

$$AFO(V, A)=<Pr^{V,A}, Fn^{V,A}; \vee, \neg, S_F^{\bar{v}}, S_P^{\bar{v}}, 'x, \exists x, =>.$$

- Such algebras form semantics base for logics
- Based on classes of such algebras various propositional, first-order, and program CNL can be defined.
- We can go further and define modal and temporal CNL.

First-Order Composition-Nominative Logics

Signature Σ: $(V, \{\vee, \neg, S_P^{\bar{v}}, S_F^{\bar{v}}, 'x, \exists x, =\}, Fs, Ps)$

Terms $Tr(V, Fs, Ps)$:

if $F \in Fs$ then $F \in Tr(V, Fs, Ps)$;

if $x \in V$ then $'x \in Tr(V, Fs, Ps)$;

if $\bar{v} = (v_1,\ldots,v_n)$ $t, t_1,\ldots,t_n \in Tr(V, Fs, Ps)$

then $S_F^{\bar{v}}(t, t_1,\ldots,t_n) \in Tr(V, Fs, Ps)$.

Formulas $Fr(V, Fs, Ps)$:

if $P \in Ps$ then $P \in Fr(V, Fs, Ps)$;

if $\Phi, \Psi \in Fr(V, Fs, Ps)$ then

$(\Phi \vee \Psi) \in Fr(V, Fs, Ps)$

$\neg\Phi \in Fr(V, Fs, Ps)$;

if $\Phi \in Fr(V, Fs, Ps)$, $\bar{v} = (v_1,\ldots,v_n)$, $t_1,\ldots, t_n \in Tr(V, Fs, Ps)$

then $S_P^{\bar{v}}(\Phi, t_1,\ldots,t_n) \in Fr(V, Fs, Ps)$;

if $x \in V$, $\Phi \in Fr(V, Fs, Ps)$ then $\exists x\Phi \in Fr(V, Fs, Ps)$;

if $t_1, t_2 \in Tr(V, Fs, Ps)$ then $t_1 = t_2 \in Fr(V, Fs, Ps)$.

CNL and Classical Logic

- FO CNL are logics of partial quasiary predicates
- Classical logic – total n-ary predicates

- Some laws of classical logic are not valued in CNL:

  - $\forall x\, P \rightarrow P$

  - modus ponens

Satisfiability Problem for FO CNL

Interpretation mappings for function and predicate symbols:

$I^{Fs}: Fs \xrightarrow{t} Fn^{V,A}, \; I^{Ps}: Ps \xrightarrow{t} Pr^{V,A}$

A triple $(AFO(V, A), \; I^{Fs}, I^{Ps})$ is called *a model for* $L(\Sigma)$.

A model is determined by *an interpretation* $J^{Fs,Ps} = (V, A, \; I^{Fs}, I^{Ps})$.

A formula Φ is called *satisfiable in an interpretation* J $(J \mid\approx \Phi)$ if there is $d \in {}^{V}A$ such that $\Phi_J(d)\downarrow = T$.

A formula Φ is called *satisfiable* $(\mid\approx \Phi)$ $J \mid\approx \Phi$ for some J

Reduction to classical first-order logic

Transform to the normal form $usnf(\Phi)$, then:

1. $clf['x] \mapsto x.$
2. $clf[S_F^{w_1,\dots,w_n}(F, t_1,\dots,t_n)] \mapsto F(clf[t_1],\dots,clf[t_n])$, $F \in Fs.$
3. $clf[(\Phi_1 \vee \Phi_2)] \mapsto (\, clf[\Phi_1] \vee clf[\Phi_2])$.
4. $clf[\neg\Phi] \mapsto \neg clf[\Phi]$.
5. $clf[S_P^{w_1,\dots,w_n}(P, t_1,\dots,t_n)] \mapsto P(clf(t_1),\dots,clf(t_n)),\ n \geq 0$.
6. $clf[\exists x\Phi] \mapsto \exists x(x \neq e \,\&\, clf[\Phi])$, $e \in U$, e is a predefined variable.
7. $clf[t_1 = t_2] \mapsto clf(t_1) = clf(t_2)$.

Theorem.

Let $\Phi \in Fr(V, Fs, Ps)$. Then $|\!\approx \Phi$ if and only if $clf[usnf[\Phi]]$ is satisfiable in the classical first-order predicate logic.

Conclusions

- CNL are based directly on program models
- CNL provide formalism for reasoning about programs
- Satisfiability problem for many classes of CNL, including first-order CNL can be reduced to the satisfiability problem for classical first-order logic
- Existent state-of-the-art methods and techniques for checking satisfiability in classical logics can also be applied to CNL
- CNL can be used in rigorous development of systems

Exercise 3

1. Solve either Variant A or Variant B of this part of the exercise:

- **Variant A:** Construct two programs Prg1 and Prg2 for evaluating $R=X^Y$ (X, Y are natural numbers): a simple one written in pure EL and a more elaborated one written in extended EL in which a function (Y *div* 2) and a predicate *odd*(Y) can additionally be used.

  As for the more elaborate version, use the technique of "Exponentiation by Squaring" (in German: "Binäre Exponentiation") which is widely described on the web.

- **Variant B:** Construct two programs Prg1 and Prg2 for evaluating $R=X*Y$ (X, Y are natural numbers): a simple one written in restricted EL (without multiplication function *) and a more elaborated one written in extended EL in which a function (Y *div* 2) and a predicate *odd*(Y) can additionally be used.

  As for the more elaborate version, use the technique of "Egyptian Multiplication" (in German: "Russische Bauernmultiplikation") which is widely described on the web.

2. Transform Prg1 and Prg2 into semantic terms STR_1 and STR_2 of EL and extended EL program algebras respectively.

3. "Test" the obtained semantic terms on 2 different states taking into account that program loops should be evaluated at least 2 times.

Exercise 4

Consider your programs formalized in Exercise 3.

1. Provide manually partial/total correctness proofs of STR_1 and STR_2 by a “traditional” mathematical method: induction on a number of loop executions.

2. Provide manually partial/total correctness proofs of STR_1 and STR_2 by using additionally Floyd-Hoare calculus specified for (E)EL program algebras.