Bull. Nov. Comp. Center, Comp. Science, 32 (2011), 107–126
© 2011 NCC Publisher

Make formal semantics easy and useful*†

N.V. Shilov

Abstract. We start with a "make easy" approach to popularize formal semantics for software engineers. It is based upon a toy language with "exoteric" operational, denotational and axiomatic semantics. Then we present a realistic and practical operational, denotational and axiomatic semantics for a simple programming language. We hope that our approach will help better education to bridge a cultural gap between Formal Methods theories and Software Engineering practice.

1. Introduction

More than 40 years have passed since Robert W. Floyd published the first research that explicitly discussed formally how to "assign meaning to programs", i.e. program formal semantics [2]. More than a decade has passed already since David A. Schmidt published an appeal *On the Need for a Popular Formal Semantics* (in ACM SIGPLAN Notices, 1997, Vol. 32). These four decades (as well as the last one) have generated number of individual formal semantics, their theories, and *more experts, but fewer general users* (due to D.A. Schmidt). Meanwhile we could see a great activity (of different kinds) in the field of formal semantics in the last decade. Let us just mention below two research on formal semantics emerged since year 2000 that attempted to comprehend the universe of formal semantics.

Peter D. Mosses promoted *The Varieties of Programming Language Semantics And Their Uses* (in particular at A.P. Ershov International Conference on Perspectives of System Informatics in 2001). He wrote (Lect. Notes Comput. Sci., Vol. 2244): "This paper surveys the main frameworks available for describing the dynamic semantics of programming languages. ... The paper is intended to be accessible to all computer scientists. Familiarity with the details of particular semantic frameworks is not required, although some understanding of the general concepts of formal semantics is assumed."

Patrik Cousot constructed a hierarchy of formal semantics in the paper *Constructive Design of a Hierarchy of Semantics of a Transition System by*

*This work is supported by the RFBR grant 09-01-00361-a.

†**Disclaimer:** This paper is not research in Computer Science or Software Engineering, but a personal experience report on teaching of a particular topic — introduction to formal semantics. Hence it does not make sense to count citations in this paper as research citations that may contribute to citation and h-indexes (so popular and valuable nowadays). Due to this reason, the paper does not provide formal references to contemporary papers, but only to the real classics of the past.

Abstract Interpretation (Theor. Comput. Sci., 2002, Vol. 277(1–2)). The hierarchy includes the maximal trace semantics of a transition system, big-step semantics, termination and nontermination semantics, Plotkin's natural, Smyth's demoniac and Hoare's angelic relational semantics and equivalent nondeterministic denotational semantics, D. Scott's deterministic denotational semantics, generalized and Dijkstra's conservative/liberal predicate transformer semantics, generalized/total and Hoare's partial correctness axiomatic semantics. In the cited paper, "all the semantics are presented in a uniform fixpoint form and the correspondences between these semantics are established through composable Galois connections, each semantics being formally calculated by abstract interpretation of a more concrete one using Kleene and/or Tarski fixpoint approximation transfer theorems."

Unfortunately, these and other attempts to comprehend the universe of formal semantics lag behind the pace of developing formal semantics. For example, in this decade a new formal semantics — so-called game semantics — has been designed and developed by Samson Abramsky and C.-H. Luke Ong. They (altogether with Dan R. Ghica and Andrzej S. Murawski) believe that the state of the art is mature for *Applying game semantics to compositional software modeling and verification* (Lect. Not. in Comp. Sci., Springer, 2004, v.2988).

Recently, David L. Parnas have called once again to "question the assumptions underlying the well-known current formal software development methods to see why they have not been widely adopted and what should be changed" in a feature article *Really Rethinking "Formal Methods"* (IEEE Computer, 2010, Vol.43, N 1).

Things are right where they started a decade ago? Not at all, since industrial applications of formal methods and formal semantics are not the unique measure of success. Let us discuss another dimension below.

A very popular (in Russia) aphorism of Mikhail (Mikhaylo) Vasilyevich Lomonosov says[1]: "Mathematics should be learned just because it disciplines the mind". This aphorism is a real classic of the past, but we do not know the exact reference: it is so popular in Russia that nobody cares about the reference. Nevertheless, there are at least two good reasons to quote the above aphorism here.

The first one is the tercentenary of M. V. Lomonosov that was celebrated November 19, 2011. Semen S. Kutateladze wrote in a recent paper *The Mathematical Background of Lomonosov's Contribution*: "Lomonosov is the Russian titan of the epoch of scientific giants" (J. App. & Ind. Math., 2011, Vol.5(2)).

The second reason is related to the educational value of Mathematics: it disciplines minds for general education. We do believe that Formal Methods

[1]translated by the authors

(and Formal Semantics in particular) discipline minds in Computer Science. We would not like to say that educators should not care about industrial applications of Formal Methods and Formal Semantics (quite the opposite, we should care!). We just want to say that a mental discipline is valuable for better education. At the same time, Formal Methods education helps to bridge the *cultural gap* [1] between Mathematics and Computer Science.

The problem is how to overcome a stable allergy to Formal Methods (Formal Semantics in particular): many people think Formal Methods are too *pure in theory* but too *poor in practice*. We do believe that the basic reason behind this allergy is the absence of *primary*, *elementary* level. It is not wise to start teaching arithmetics from Peano axiomatic, but it is common sense to start from elementary problems about the number of apples, pencils, etc. For example, nobody teaches first graders to prove in Peano axiomatic

$$\vdash \forall x.\forall y.\forall z: \ ((x+y)+z) = (x+(y+z)),$$

but everyone teaches to solve elementary problems like the following one:

> I gave 5 apples to Peter, and he gave 2 apples to John. How many apples did Peter have after that?

(If you think that he had 3 apples, you are not right, since he had 3 *at least.*)

In our vision, part of the reason why there is no demand for Formal Methods among students and engineers is that FM-experts do not care about primary education in this field at an early stage of higher education. In particular, many courses on Formal Semantics start with fearful terms like *state machine*, *predicate transformer*, *logic inference*, *operational semantics*, *denotational semantics*, *axiomatic semantics* without *elementary* explanations of the basic notions. We would like to present some examples of elementary explanations of basic notions in the following sections. They were used in one-semester course for the first-year students *Programming-I* (*Introduction to Computer Science and Programming Languages*) at the Mathematics and Mechanics Department of Novosibirsk State University in 2008-2010.

It is worth to remark that there is a trend in Formal Methods community to popularize individual Formal Methods via puzzles, competitions and challenging contests. For example, a decade ago Kwangeun Yi and the author of the present paper published the article *Engaging Students with Theory through ACM Collegiate Programming Contests* (CACM, 2002, v.45(9)) and then the introductive paper *How to find a coin: propositional program logics made easy* (Current Trends in Theoretical Computer Science by World Scientific, 2004, v.2). Recently, Yury G. Karpov has published a comprehensive textbook (in Russian) *Model checking: Verification of parallel and distributed program systems* (BHV-Petersburg, 2010), where a variety of

```
⟨program⟩ ::= ⟨assignment⟩ | (⟨program⟩) | ⟨program⟩ ; ⟨program⟩
|
                  | if ⟨condition⟩ then ⟨program⟩ else ⟨program⟩ |
                                   | while ⟨condition⟩ do ⟨program⟩
⟨assignment⟩ ::= ⟨variable⟩:= ⟨expression⟩
⟨condition⟩ ::= ⟨(in)equality⟩ | (⟨condition⟩) | ¬⟨condition⟩ |
        | ⟨condition⟩ ∧ ⟨condition⟩ | ⟨condition⟩ ∨ ⟨condition⟩
⟨(in)equality⟩ ::= ⟨expression⟩ = ⟨expression⟩ |
   | ⟨expression⟩ < ⟨expression⟩ | ⟨expression⟩ ≤ ⟨expression⟩
|
   | ⟨expression⟩ > ⟨expression⟩ | ⟨expression⟩ ≥ ⟨expression⟩
⟨expression⟩ ::= ⟨constant⟩ | ⟨variable⟩ | (⟨expression⟩) |
 | ⟨expression⟩ + ⟨expression⟩ | ⟨expression⟩ − ⟨expression⟩ |
                                   | ⟨expression⟩ ∗ ⟨expression⟩
```

Figure 1. BNF definition of the ToL syntax

puzzles are solved by encoding them in SPIN model checker. Also, there are some other examples. And yet we think that this educational approach should be preceded by much more elementary examples of formal methods, since nowadays too many students are weak in mathematics.

2. A toy language ToL

A programming language is any artificial computer language designed for the organization of automatic data processing, i.e. data and process representation, handling and management. For natural and artificial languages (including computer languages), the terms *syntax*, *semantics* and *pragmatics* are used to categorize descriptions of language characteristics. The syntax is the orthography of the language. The meaning of syntactically-correct constructs is provided through the language semantics. Pragmatics is the practice of using the meaningful syntactically-correct constructs.

2.1. Formal syntax, informal semantics, and pragmatics of ToL

A language ToL (Toy Language) is described below. It is not a *programming language* as specified in the above paragraph, but it looks like a programming language due to its syntax.

The definition of the ToL syntax is given in Figure 1. (It assumes acquaintance with Backus-Naur notation.) Variables and constants can be defined in the standard manner: the former are identifiers, the latter are numbers (integer or rational) in any number system. An example of a correct ToL program is given in Figure 2; let us refer to this program as the sample program SP throughout the paper.

```
if z<0 then z:= -1
       else (x:= 0 ; y:= 0 ;
             while y≤z do (y:= y + 2*x + 1 ; x:= x + 1) ;
             x:= x - 1)
```

Figure 2. A sample ToL program SP

Semantics of the language is non-conventional and non-programming it is artificial. It can be defined informally as follows: semantics of a ToL program is an integer that is the length (the number of assignments) in the shortest path throughout the program flowchart (i.e. from *start* to *end*). (For example, semantics of the above sample ToL program is 1.) This semantics is not for program execution or verification, but for illustrating and explaining what is *semantics* and what *kinds* of semantics are in use. Nevertheless, let us continue to call “programs” all syntactically correct words of ToL (in spite of non-conventional non-programming semantics).

Thus ToL pragmatics is to illustrate and explain what are *operational*, *denotational* and *axiomatic* semantics, what good properties they have (but not why they are good), and what is the nature of Formal Semantics.

2.2. Ways to assign meanings to ToL

2.2.1. Operational semantics: a machine that computes.

Operational semantics *translates* programs to corresponding *machines* (of a certain class) whose operations are (conventionally) “executable”: semantics of a program is defined, in a way, by all admissible executions of the corresponding machine.

In the case of ToL, the target class of machinery consists of recursive algorithms without input that compute integer values; a recursive algorithm that corresponds to a program α (and the value that it computes) is denoted by F_α; the translation (the correspondence) is defined as follows:

1. $F_{\langle\texttt{assignment}\rangle} = 1$;
2. $F_{(\beta)} = F_\beta$;
3. $F_{\beta;\gamma} = F_\beta + F_\gamma$;
4. $F_{\texttt{if}\ \langle\texttt{condition}\rangle\ \texttt{then}\ \beta\ \texttt{else}\ \gamma} = \min\{F_\beta, F_\gamma\}$;
5. $F_{\texttt{while}\ \langle\texttt{condition}\rangle\ \texttt{do}\ \beta} = 0$.

In particular, the operational semantics of the sample program SP is

$$F_{\texttt{if z<0 then z:=-1 else(x:=0;y:=0; while y}\leq\texttt{z do(y:=y+2*x+1;x:=x+1);x:=x-1)}} =$$
$$= \min\{F_{\texttt{z:=-1}},\ F_{\texttt{x:=0;y:=0; while y}\leq\texttt{z do(y:=y+2*x+1;x:=x+1);x:=x-1}}\} =$$
$$= \min\{1,\ F_{\texttt{x:=0}} + F_{\texttt{y:=0; while y}\leq\texttt{z do(y:=y+2*x+1;x:=x+1);x:=x-1}}\} =$$

$$= \min\{1,\ 1 + F_{\texttt{y:=0}} + F_{\texttt{while y}\leq\texttt{z do(y:=y+2*x+1;x:=x+1);x:=x-1}}\} = \\ = \min\{1,\ 1+1+\dots\} = 1,$$

which can be routinely computed in full details.

Let us remark that as soon as a formal semantics is provided for a language, it induces an equivalence relation on programs. For example, ToL programs α and β are said to be equivalent if they have equal semantics, i.e. $F_\alpha = F_\beta$. Of course, if a language has several semantics, then these semantics can induce different (disjoint maybe) equivalences.

2.2.2. Denotational semantics: an algebra for calculations.

An *algebra* is a set of objects with operations on them. For example, natural numbers $\mathcal{N}$ with operations '0' and '1' of zero-arity (constants), binary operations '+' (i.e. $\lambda x, y.(x + y)$) and '−' (i.e. $\lambda x, y. \max\{0, (x - y)\}$) form an algebra. The same set $\mathcal{N}$ with a constant '0', unary operations '+1' (i.e. $\lambda x.(x+1)$) and 'I' (for Identical function $\lambda x.x$), and a binary operation 'min' (i.e. $\lambda x, y. \min\{x, y\}$) form another algebra, since it uses a different set of available operations.

Denotational semantics assigns (in a consistent compositional manner) the elements of some algebra to programs and the operations of this algebra to program constructs; usually the assigning function is denoted by $[\![\ ]\!]$.

For example, let us consider the following algebra $\langle\mathcal{N},\ 0,\ 1,\ \mathrm{I},\ +,\ \min\rangle$ and the following assignment of its elements to ToL programs and its operations to ToL program constructs ('`(...)`', '`;`', '`if-then...else...`', '`while-do...`'):

1. $[\![\langle\texttt{assignment}\rangle]\!] = 1$;
2. $[\![(...)]\!] = \mathrm{I}$;
3. $[\![;]\!] = + \equiv \lambda x, y.(x + y)$;
4. $[\![\texttt{if} - \texttt{then...else...}]\!] = \min \equiv \lambda x, y. \min\{x, y\}$;
5. $[\![\texttt{while} - \texttt{do...}]\!] = 0$;
6. $[\![\texttt{prog_constr}(\alpha, \beta, \dots)]\!] = [\![\texttt{prog_constr}]\!]([\![\alpha]\!], [\![\beta]\!], \dots)$ for every $\texttt{prog_constr} \in \{\ (...),\ ;\ ,\ \texttt{if} - \texttt{then...else...}, \texttt{while} - \texttt{do...}\}$.

In particular, the denotational semantics of the sample program `SP` can be calculated in the algebra $\langle\mathcal{N},\ 0,\ 1,\ +,\ \min\rangle$ as follows:

```
[[if z<0 then z:= -1
        else (x:= 0 ; y:= 0 ;
              while y≤z do (y:= y + 2*x + 1 ; x:= x + 1) ;
              x:= x - 1)]] =
= [[if - then...else...]]
          ([[z:= -1]] , [[x:= 0 ; y:= 0 ;
```

```
                    while y≤z do (y:= y + 2*x + 1 ; x:= x + 1)
;
                    x:= x - 1]]) =
= min{ 1 , [[;]]([[x:= 0]] , [[y:= 0 ;
                    while y≤z do (y:= y + 2*x + 1 ; x:= x + 1)
;
                    x:= x - 1]])} =
= min{ 1 , 1 +  [[;]]([[y:= 0]] ,
                  [[while y≤z do (y:= y + 2*x + 1 ; x:= x + 1)
;
                     x:= x - 1]])} =
= min{ 1 , 1 + (1 + ...)} = 1.
```

Let us remark that denotational semantics assigns meanings to programs as well as to program constructs (in contrast to operational semantics).

Operational and denotational semantics of ToL are closely related.

Statement 1. $F_\alpha = [\![\alpha]\!]$ *for every ToL program* α*.*

Proof hint: induction on the program structure. ■

Let us refer to the property stated in the above statement 1 as *(mutual) soundness* of operational and denotational semantics of ToL. Soundness implies that operational and denotational ToL semantics induce one program equivalence. In the general case, relations between operational and denotational semantics can be much more complicated.

2.2.3. Axiomatic Semantics: code-driven proof.

An axiomatic system is a calculus, i.e. a set of syntactic *inference rules* for deriving ("proving") new "facts" (that are called *theorems*) from *axioms* (i.e. inference rules without premises).

Axiomatic semantics for ToL is an axiomatic system for *assertions* of the following form ⟨`constant`⟩ ≤ ⟨`program`⟩ ≤ ⟨`constant`⟩, where each constant is a non-negative integer or infinity ∞, but the former should not be greater than the latter.

ToL axiomatic semantics comprises the axioms and inference rules presented in Table 1. An example of derivation in this system of the assertion $1 \leq$ `SP` ≤ 1 for the sample program is depicted in Figure 3.

An assertion $m \leq \alpha \leq n$ is said to be valid, if $m \leq [\![\alpha]\!] \leq n$ (or, according to the soundness of ToL semantics, $m \leq F_\alpha \leq n$). Axiomatic semantics is said to be sound if every derivable assertion is valid; if every valid assertion is derivable, then semantics is said to be complete.

Statement 2. *Tol axiomatic semantics is sound and complete.*

Table 1. ToL axiomatic semantics

Assignment axiom: $\frac{}{1\leq\langle\texttt{assignment}\rangle\leq 1}$	Loop axiom: $\frac{}{0\leq\texttt{while}...\texttt{do}...\leq 0}$
Block rule: $\frac{m\leq\alpha\leq n}{m\leq(\alpha)\leq n}$	Stretching rule: $\frac{m'\leq\alpha\leq n'}{m\leq\alpha\leq n}$, $m \leq m'$, $n' \leq n$
Composition rule: $\frac{m'\leq\alpha\leq n' \qquad m''\leq\beta\leq n''}{m\leq\alpha;\beta\leq n}$, $m = m' + m''$, $n = n' + n''$	
Then rule: $\frac{m\leq\alpha\leq n \qquad m\leq\beta\leq\infty}{m\leq\texttt{if}...\texttt{then}\ \alpha\ \texttt{else}\ \beta\leq n}$	Else rule: $\frac{m\leq\alpha\leq\infty \qquad m\leq\beta\leq n}{m\leq\texttt{if}...\texttt{then}\ \alpha\ \texttt{else}\ \beta\leq n}$

Proof hint: induction on the program structure (completeness) and induction on the length (height) of derivation (soundness). ■

In particular, it is not a surprise that a valid assertion $1 \leq \texttt{SP} \leq 1$ is proveable in the axiomatic semantics, as we have seen in Figure 3.

3. Making Toy Programming Language ToyPL

As we have already stated explicitly, the toy language ToL is not a programming language at all, but a language where correct words just look like programs. Now we want to make a programming language with the same syntax as ToL by providing a *programming* semantics for it. As a result, we will get another language that we would refer to as Toy Programming Language ToyPL: its syntax is the same, but ToyPL and ToL are different languages since they have different semantics. Here we are speaking not about a set of closely related semantics of different kinds (like operational, denotational and axiomatic semantics for ToL), but disjoint semantics of similar kinds. Thus pragmatics of ToyPL is to illustrate a programming semantics of operational, denotational and axiomatic type.

Since a programming language is a language for organizing automatic data processing, the best way to represent programming language is to describe its *implementation semantics*, how it works on a "computer", and what processes of data transformation are defined by its programs on this platform. Since we have no any particular target platform in mind, we have to define the ToyPL *virtual machine*.

A virtual machine is an abstract "computer" with an instruction set executable (interpretable) at any conventional computer platform. Mendel

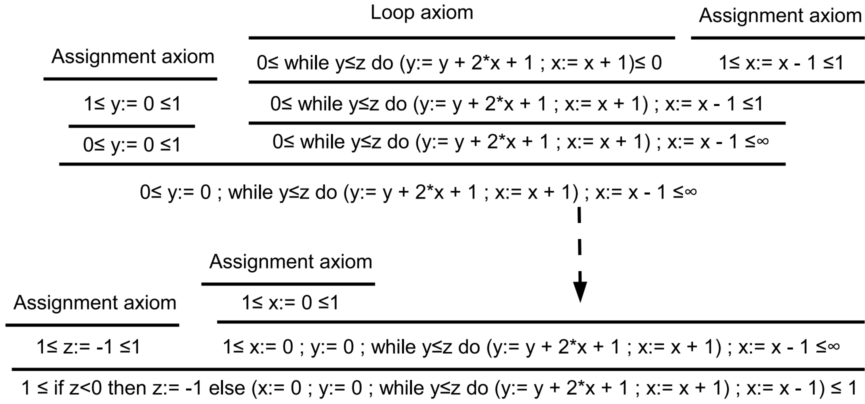

Figure 3. An example of an inference in ToL

Rosenblum wrote (in ACM Queue, 2004, v.2):

> The term virtual machine initially described a 1960s operating system concept: a software abstraction with the looks of a computer system's hardware (real machine). Forty years later, the term encompasses a large range of abstractions — for example, Java virtual machines that don't match an existing real machine. Despite the variations, in all definitions the virtual machine is a target for a programmer or compilation system. In other words, software is written to run on the virtual machine.

The instruction set of ToyPL virtual machine (ToyPL-VM in the sequel) consists of (labeled) operators of the following two kinds:

- assignment "⟨`label`⟩`:` ⟨`variable`⟩`:=` ⟨`expression`⟩ `goto` ⟨`label`⟩`;`"
- and choice "⟨`label`⟩`:` ⟨`condition`⟩ `then` ⟨`label`⟩ `else` ⟨`label`⟩`;`"

where labels are natural numbers (including 0) in any fixed numeric system (decimal hereafter). A program (or a *byte-code*) of/for ToyPL-VM is a finite set of operators marked by disjoint labels. A sample program for ToyPL-VM is presented in Figure 4; let us refer to this program `TP` throughout the paper.

Let $N \geq 1$ be some fixed integer parameter that characterizes the bit-size of our virtual machine's word. Assume that S is a program of ToyPL-VM

```
0:  if z<0 then 1 else 2;
1:  z:= -1 goto 8;
2:  x:= 0 goto 3;
3:  y:= 0 goto 4;
4:  if y≤z then 5 else 7;
5:  y:= y+2*x+1 goto 6;
6:  x:= x+1 goto 4;
7:  x:= x-1 goto 8;
```

Figure 4. A sample program `TP` for ToyPL-VM

and let $VAR(S)$ be the set of all variables that occur (have instances) in S. A *state* (of S) is a total function $s : VAR(S) \to \mathbb{Z}/2^N$ that assigns every variable $x \in VAR(S)$ some residuals in $\mathbb{Z}/2^N$, i.e. an integer number $s(x) \in [0...(2^N - 1)]$ that is called the value of x in the state s. A configuration (of S) is a pair of the form *(label, state)*, where *label* has instance(s) in S (i.e. marks any operator, or occurs after `goto`, `then` or `else` in some operator) and *state* is a state[2].

Let l`:` x`:=` t `goto` l'`;` be an assignment within S (i.e. l and l' are labels, x is a variable, and t is an expression). A *firing* of the assignment is any pair of configurations (l, s), (l', s'), where s and s' are states such that[3] $s' = upd(s, x, s(t))$, i.e. for every variable y within S the following holds:

$$s'(y) =$$
$$\begin{cases} s(y), \text{ if } y \not\equiv x \text{ (i.e. } y \text{ is not } x), \\ s(t), \text{ i.e. the value of the expression } t \text{ in the state } s, \text{if } y \equiv x \text{ (i.e. } y \text{ is } x), \end{cases}$$

where all arithmetic computations (in t) are carried out modulo 2^N.

Let l`:` ϕ `then` l^+ `else` l^- be a choice within S (i.e. l, l^+, and l^- are labels, ϕ is a condition). A *firing* of the choice is any pair of configurations (l, s), (l', s), where s is a state and l' is a label in $\{l^+, l^-\}$ such that

$$l' = \begin{cases} l^+, \text{ if } s \models_{\mathbb{Z}/2^N} \phi \text{ (i.e. } \phi \text{ holds in the state } s), \\ l^-, \text{ if } s \not\models_{\mathbb{Z}/2^N} \phi \text{ (i.e. } \phi \text{ does not hold in the state } s), \end{cases}$$

where all arithmetic computations (in ϕ) are carried out modulo 2^N and then the resulting values are compared as natural numbers[4].

Let a *computational step* (of S) be a firing of any assignment or choice within this program. Let a *computation* (of S) be any finite or infinite sequence of configurations $(l_0, s_0), \dots (l_i, s_i), , (l_{(i+1)}, s_{(i+1)}), \dots$ such that every pair of consequent configurations $(l_i, s_i), (l_{(i+1)}, s_{(i+1)})$ within this sequence is a computational step of S; the computation is said to be *complete*

[2]We assume that ToyPL-VM has disjoint memory for programs and data: labels can be natural numbers, while variable values are residuals modulo 2^N.

[3]Hereafter *upd* stays for *update*.

[4]It implies that a programmer should be in charge for overflow.

$(0,\ 10,15,5)$, $(2,\ 10,15,5)$, $(3,\ 0,15,5)$, $(4,\ 0,0,5)$,
$(5,\ 0,0,5)$, $(6,\ 0,1,5)$, $(4,\ 1,1,5)$, $(5,\ 1,1,5)$, $(6,\ 1,4,5)$,
$(4,\ 2,4,5)$, $(5,\ 2,4,5)$, $(6,\ 2,9,5)$, $(4,\ 3,9,5)$, $(7,\ 3,9,5)$, $(8,\ 2,9,5)$

Figure 5. An example of a complete computation of `TP` in case $N = 4$.

or to be a *run* (of S) if it starts with the label $l_0 \equiv 0$ and is either infinite or finishes with any *terminal* label l_n (i.e. does not mark any operator within the program).

An example of a finite run of the program `TP` is presented on Figure 5. In this example, we assume that $N = 4$ and configurations are represented by quadruples as follows: a configuration (l, s), where l is a label and $s : \{\mathtt{x}, \mathtt{y}, \mathtt{z}\} \to \mathbb{Z}/2^N$ is a state, is represented by the vector $(l,\ s(\mathtt{x}), s(\mathtt{y}), s(\mathtt{z}))$.

Implementation semantics for ToyPL relies upon the following *translation* algorithm TR that maps every ToyPL-program α to the corresponding program S_α for ToyPL virtual machine. The idea behind this algorithm is quite trivial: draw a flowchart of a ToyPL-program, enumerate operators in the right order, use these numbers as labels and represent the enumerated operators as labeled operators of the corresponding program of ToyPL-VM. But formally the algorithm is defined recursively by the program structure as presented in Figure 6. It uses the following auxiliary notation:

- If S is a program for ToyPL-VM, then let $\max(S)$ be the maximal label (i.e. a natural number) that has an instance in S.
- If S_1 and S_2 are programs for ToyPL-VM without labels that mark operators in S_1 and S_2 simultaneously, then let $S_1 \cup S_2$ be a program for ToyPL-VM that comprises both sets of operators S_1 and S_2.
- If S is a program for ToyPL-VM and k is a label (i.e. a natural number), then $(S + k)$ is a program for ToyPL-VM that results from S by instantiating an integer number $(l + k)$ instead of every instance of every label l.
- If S is a program for ToyPL-VM and l and k are labels, then $S(l/k)$ is a program for ToyPL-VM that results from S by instantiating the label l instead of every instance of the label k.

This algorithm has the following very nice structural property.

Statement 3. *For every ToyPL-program α, the set of labels in $TR(\alpha)$ is the interval $[0..\max(TR(\alpha))]$ and $\max(TR(\alpha))$ is the unique terminal label in $TR(\alpha)$.*

- For any variable x and expression t, let $TR(x{:=}t)$ be `0` : $x{:=}t$ `goto 1`;
- For every ToyPL-program α, let $TR((\alpha)) = TR(\alpha)$.
- For all ToyPL-programs α and β let $TR(\alpha\ ;\ \beta)$ be
$$TR(\alpha)\ \cup\ (TR(\beta) + \max(TR(\alpha))).$$
- For all ToyPL-programs α and β and every condition ϕ let $TR(\texttt{if}\ \phi\ \texttt{then}\ \alpha\ \texttt{else}\ \beta)$ be `0:` `if` ϕ `then 1 else` $(1 + \max(TR(\alpha)))$`;` $\cup$
$$\cup\ (TR(\alpha)((\max(TR(\alpha)) + \max(TR(\beta)))/\max(TR(\alpha))) + 1) \cup$$
$$\cup\ (TR(\beta) + \max(TR(\alpha)) + 1).$$
- For every ToyPL-programs α and every condition ϕ let $TR(\texttt{while}\ \phi\ \texttt{do}\ \alpha)$ be
`0:` `if` ϕ `then 1 else`
$1 + \max(TR(\alpha))$; $\cup$ $(TR(\alpha) + 1)(0/(1 + \max(TR(\alpha))))$.

Figure 6. TR: a recursive translation to byte-code

.

Proof hint: induction on the program structure. Please see for details a new Russian textbook *Introduction to Parsing, Semantics, Compilation and Verification of Programs*, published by the author in 2011 at Novosibirsk State University. ■

There is no room to do a routine exercise, but let us remark that $TR(\texttt{SP}) =$ TP. (For the details please refer to the textbook *Introduction to Parsing, Semantics, Compilation and Verification of Programs.*)

The implementation semantics is a particular instance of the operational semantics where the underling machine is a virtual computer. In particular, let us define the *implementation semantics* of ToyPL (which makes it a *programming language*) as follows: for every ToyPL-program α let the implementation semantics of α be the set of all runs of the translated program $TR(\alpha)$ on ToyPL-VM. For the first time the implementation semantics for imperative programming languages was introduced as part of the so-called Vienna Development Method (VDM), one of the longest-established Formal Methods. VDM has grown at IBM's Vienna Laboratory in the 1970s under the supervision of Dines Bjørner and Cliff Jones (*The Vienna Development Method: The Meta-Language*, Lect. Not. in Comp. Sci., v.61, 1978.)

For example, the implementation semantics of the sample program SP (considered as a ToyPL-program hereafter) is equal to the set of all possible runs of the program TP. This set consists of three disjoint parts:

- all finite sequences of configurations that have the following form
$(0,\ p, q, r)$, $(2,\ p, q, r)$, $(3,\ 0, q, r)$, $(4,\ 0, 0, r)$,

$(5,\ 0,0,r)\ ,\ (6,\ 0,1,r)\ ,\ (4,\ 1,1,r)\ ,\ \ldots\ldots\ldots\ (4,\ \lfloor\sqrt{r}\rfloor,\lfloor\sqrt{r}\rfloor^2,r)\ ,$
$(5,\ \lfloor\sqrt{r}\rfloor,\lfloor\sqrt{r}\rfloor^2,r)\ ,\ (6,\ \lfloor\sqrt{r}\rfloor,(\lfloor\sqrt{r}\rfloor+1)^2,r)\ ,\ (4,\ (\lfloor\sqrt{r}\rfloor+1),(\lfloor\sqrt{r}\rfloor+1)^2,r)\ ,$
$(7,\ (\lfloor\sqrt{r}\rfloor+1),(\lfloor\sqrt{r}\rfloor+1)^2,r)\ ,\ (8,\ \lfloor\sqrt{r}\rfloor,(\lfloor\sqrt{r}\rfloor+1)^2,r),$
where $(\lfloor\sqrt{r}\rfloor+1)^2 < 2^N$;

- all finite sequences of configurations that have the following form
  $(0,\ p,q,r)\ ,\ (2,\ p,q,r)\ ,\ (3,\ 0,q,r)\ ,\ (4,\ 0,0,r)\ ,$
  $(5,\ 0,0,r)\ ,\ (6,\ 0,1,r)\ ,\ (4,\ 1,1,r)\ ,\ \ldots\ldots\ldots\ (4,\ \lfloor\sqrt{r}\rfloor,\lfloor\sqrt{r}\rfloor^2,r)\ ,$
  $(5,\ \lfloor\sqrt{r}\rfloor,\lfloor\sqrt{r}\rfloor^2,r)\ ,$
  $(6,\ \lfloor\sqrt{r}\rfloor,(\lfloor\sqrt{r}\rfloor+1)^2 mod(2^N),r)\ ,$
  $(4,\ (\lfloor\sqrt{r}\rfloor+1)mod(2^N),(\lfloor\sqrt{r}\rfloor+1)^2 mod(2^N),r)\ ,\ \ldots$
  $\ldots\ (5,\ (\lfloor\sqrt{r}\rfloor+k)mod(2^N),(\lfloor\sqrt{r}\rfloor+k)^2 mod(2^N),r)\ ,$
  $(6,\ (\lfloor\sqrt{r}\rfloor+k)mod(2^N),(\lfloor\sqrt{r}\rfloor+k+1)^2 mod(2^N),r)\ ,$
  $(4,\ (\lfloor\sqrt{r}\rfloor+k+1)mod(2^N),(\lfloor\sqrt{r}\rfloor+k+1)^2 mod(2^N),r)\ ,$
  $(7,\ (\lfloor\sqrt{r}\rfloor+k+1)mod(2^N),(\lfloor\sqrt{r}\rfloor+k+1)^2 mod(2^N),r)\ ,$
  $(8,\ (\lfloor\sqrt{r}\rfloor+k)mod(2^N),(\lfloor\sqrt{r}\rfloor+k+1)^2 mod(2^N),r),$
  where $(\lfloor\sqrt{r}\rfloor+1)^2 mod(2^N) \le r$ but $(\lfloor\sqrt{r}\rfloor+k+1)^2 mod(2^N) > r$ for some $k > 0$;
- all infinite sequences of configurations that have the following form
  $(0,\ p,q,r)\ ,\ (2,\ p,q,r)\ ,\ (3,\ 0,q,r)\ ,\ (4,\ 0,0,r)\ ,$
  $(5,\ 0,0,r)\ ,\ (6,\ 0,1,r)\ ,\ (4,\ 1,1,r)\ ,\ \ldots\ldots\ldots\ (4,\ \lfloor\sqrt{r}\rfloor,\lfloor\sqrt{r}\rfloor^2,r)\ ,$
  $(5,\ \lfloor\sqrt{r}\rfloor,\lfloor\sqrt{r}\rfloor^2,r)\ ,$
  $(6,\ \lfloor\sqrt{r}\rfloor,(\lfloor\sqrt{r}\rfloor+1)^2 mod(2^N),r)\ ,$
  $(4,\ (\lfloor\sqrt{r}\rfloor+1)mod(2^N),(\lfloor\sqrt{r}\rfloor+1)^2 mod(2^N),r)\ ,\ \ldots$
  $\ldots\ (5,\ (\lfloor\sqrt{r}\rfloor+k)mod(2^N),(\lfloor\sqrt{r}\rfloor+k)^2 mod(2^N),r)\ ,$
  $(6,\ (\lfloor\sqrt{r}\rfloor+k)mod(2^N),(\lfloor\sqrt{r}\rfloor+k+1)^2 mod(2^N),r)\ ,$
  $(4,\ (\lfloor\sqrt{r}\rfloor+k+1)mod(2^N),(\lfloor\sqrt{r}\rfloor+k+1)^2 mod(2^N),r)\ ,\ \ldots\ ,$
  where $(\lfloor\sqrt{r}\rfloor+k)^2 mod(2^N) \le r$ for all $k > 0$.

Here p, q and r are integers in the range $[0..(2^N-1)]$, and $\lfloor\ldots\rfloor$ denotes the *floor function* (which returns the largest integer not greater than the argument value).

The above example demonstrates *formally* that the ToyPL-program `SP` and the corresponding program `TP` for ToyPL-VM either compute (in the variable `x`) the integer part of the square root of the initial value r (of the variable `z`) in the case when $(\lfloor\sqrt{r}\rfloor+1)^2 < r$, or diverge in the opposite case.

Since the implementation semantics for ToyPL is already defined, we can define *input-output* semantics for ToyPL-programs as follows: for every ToyPL-program α and all states[5] s' and s'' of α, let us write $(s',s'') \in IO(\alpha)$ and say that α outputs s'' for input s', if there exists a run of $TR(\alpha)$

[5]We have defined a state for programs of ToyPL-VM as a total function that maps every variable into its value. The same definition holds for ToyPL-programs.

Table 2. ToyPL axiomatic semantics

Assignment axiom: $\frac{}{\{\psi_{t/x}\}x{:=}t\{\psi\}}$, where $\psi_{t/x}$ denotes the result of substitution of t into ψ instead of all *free* instances of x	
Stretching rule: $\frac{\{\phi'\}\alpha\{\psi'\}}{\{\phi\}\alpha\{\psi\}}$, $\models_{\mathbb{Z}/2^N} (\phi \to \phi')$ and $\models_{\mathbb{Z}/2^N} (\psi' \to \psi)$	
Block rule: $\frac{\{\phi\}\alpha\{\psi\}}{\{\phi\}(\alpha)\{\psi\}}$	Composition rule: $\frac{\{\phi\}\alpha\{\xi\} \qquad \{\xi\}\beta\{\psi\}}{\{\phi\}(\alpha\ ;\ \beta)\{\psi\}}$
IF rule: $\frac{\{\phi\wedge\xi\}\alpha\{\psi\} \qquad \{\phi\wedge\neg\xi\}\beta\{\psi\}}{\{\phi\}\texttt{if}\ \xi\ \texttt{then}\ \alpha\ \texttt{else}\ \beta\{\psi\}}$	WHILE rule: $\frac{\{\iota\wedge\xi\}\alpha\{\iota\}}{\{\iota\}\texttt{while}\ \xi\ \texttt{do}\ \alpha\{\iota\wedge\neg\xi\}}$

that starts with s' and finishes with s''. For example, `SP` always outputs $((\lfloor\sqrt{r}\rfloor + k)mod(2^N), (\lfloor\sqrt{r}\rfloor + k + 1)^2 mod(2^N), r)$, where $k \geq 0$ is the first integer such that $(\lfloor\sqrt{r}\rfloor + k)^2 mod(2^N) \leq r$ but $(\lfloor\sqrt{r}\rfloor + k + 1)^2 mod(2^N) > r$.

The axiomatic semantics for ToyPL is an axiomatic system for *assertions* of the following form $\{\langle$`pre-condition`$\rangle\}\langle$`program`$\rangle\{\langle$`post-condition`$\rangle\}$, where both conditions are first-order formulas in the signature of $\mathbb{Z}/2^N$. ToyPL axiomatic semantics comprises the axioms and inference rules presented in Table 2. The axiomatic semantics for imperative programming languages was introduced by Charles Antony Richard Hoare in the paper *An Axiomatic Basis for Computer Programming* (Comm. ACM, 1969, v.12, n.10); maybe, this axiomatic semantics was the first ever formal semantics for programming languages (while it relied upon informal implementation semantics at the moment).

Let us say that an assertion $\{\phi\}\alpha\{\psi\}$ is *valid* and write $\models_{\mathbb{Z}/2^n} \{\phi\}\alpha\{\psi\}$ if for all states s' and s'', $s' \models_{\mathbb{Z}/2^n} \phi$ and $(s', s'') \in IO(\alpha)$ implies $s'' \models_{\mathbb{Z}/2^n} \psi$.

Statement 4. *ToyPl axiomatic semantics is sound and complete.*

Proof (sketch). Soundness can be proved in a routine manner by induction on the length (height) of a derivation. Completeness can be proved by induction on the program structure and use of *the weakest pre-conditions*: a formula ϕ is the weakest pre-condition for a program α and a post-condition ψ, if the following holds:

- $\models_{\mathbb{Z}/2^N} \{\phi\}\alpha\{\psi\}$,
- $\models_{\mathbb{Z}/2^N} (\phi' \to \phi)$ for every formula ψ' such that $\models_{\mathbb{Z}/2^N} \{\phi'\}\alpha\{\psi\}$.

Table 3. ToyPL structural operational semantics ToyPL-SOS

Assignment axiom: $\frac{}{s\langle x{:=}t\rangle upd(s,x,s(t))}$	Loop axiom: $\frac{}{s\langle \texttt{while}\ \phi\ \texttt{do}\ \alpha\rangle s}$, if $s \not\models_{\mathbb{Z}/2^N} \phi$
Block rule: $\frac{s'\langle\alpha\rangle s''}{s'\langle(\alpha)\rangle s''}$	Composition rule: $\frac{s'\langle\alpha\rangle s'' \qquad s''\langle\beta\rangle s'''}{s'\langle\alpha;\beta\rangle s'''}$
Loop rule: $\frac{s'\langle\alpha\rangle s'' \qquad s''\langle \texttt{while}\ \phi\ \texttt{do}\ \alpha\rangle s'''}{s'\langle \texttt{while}\ \phi\ \texttt{do}\ \alpha\rangle s'''}$, if $s' \models_{\mathbb{Z}/2^N} \phi$	
Then rule: $\frac{s'\langle\alpha\rangle s''}{s'\langle \texttt{if}\ \phi\ \texttt{then}\ \alpha\ \texttt{else}\ \beta\rangle s''}$, if $s \models_{\mathbb{Z}/2^N} \phi$	
Else rule: $\frac{s'\langle\beta\rangle s''}{s'\langle \texttt{if}\ \phi\ \texttt{then}\ \alpha\ \texttt{else}\ \beta\rangle s''}$, if $s \not\models_{\mathbb{Z}/2^N} \phi$	

The weakest pre-condition for every postcondition and every program can be constructed due to the following arguments: since N, the size of the ToyPL-VM word, is some fixed integer number, the states-space of every ToyPL program is finite; hence every set of states of every ToyPL program is finite and can be specified by an appropriate first-order formula. With the aid of the weakest pre-conditions, we can proceed with the completeness proof like in the textbook *Introduction to Parsing, Semantics, Compilation and Verification of Programs.* ■

4. ToyPL structural operational semantics

Structural Operational Semantics (SOS) was introduced by Gordon D. Plotkin in the technical report *A Structural Approach to Operational Semantics* ((Comp. Sci. Dep. of Aarhus University, Denmark, 1981). It is usually presented in a form of an axiomatic system and assumes some *semantic inference machine* driven by the program structure.

In particular, structural operational semantics for ToyPL is presented in Table 3. This axiomatic system is designed for reasoning about triples of the form $s'\langle\alpha\rangle s''$, where s' and s'' are states, and α is a ToyPL-program.

An example of an inference in ToyPL-SOS is presented in Figure 7. In this example, we assume that $N = 4$ and states are represented by triples as follows: a state $s : \{\mathtt{x}, \mathtt{y}, \mathtt{z}\} \to \mathbb{Z}/2^N$ is represented by the vector

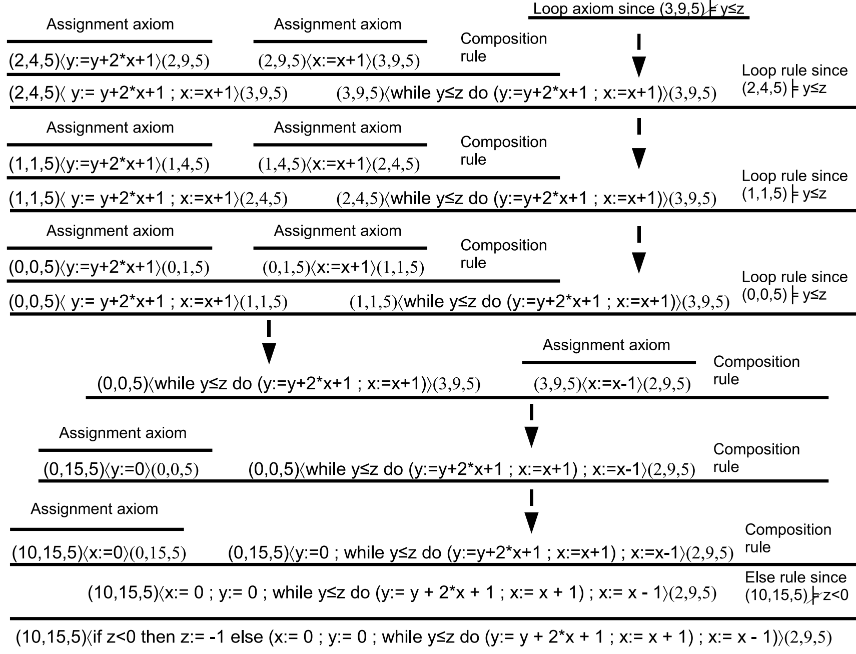

Figure 7. An example of ToyPL-SOS inference

$(s(\mathtt{x}), s(\mathtt{y}), s(\mathtt{z}))$.

Statement 5. *For every ToyPL-program α and all states s' and s'', the following holds: $(s', s'' \in IO(\alpha) \;\Leftrightarrow \vdash_{ToyPL} s'\langle\alpha\rangle s''$ (or, in words, α outputs s'' for input s' iff triple $s'\langle\alpha\rangle s''$ is provable in ToyPL structural axiomatic semantics.)*

Proof hint: $\Rightarrow$-direction — induction on the program structure, $\Leftarrow$-direction — induction on height of the inference tree. Please see for details the textbook *Introduction to Parsing, Semantics, Compilation and Verification of Programs.* ■

Thus structural operational semantics of ToyPL is sound and complete. It implies that for all integers $p, q, r, p', q', r' \in [0..(2^N - 1)]$ the following holds: $\vdash_{ToyPL} (p, q, r)\langle\mathtt{SP}\rangle(p', q', r')$ iff $p' = (\lfloor\sqrt{r}\rfloor + k)mod(2^N)$, $q' = (\lfloor\sqrt{r}\rfloor + k+1)^2 mod(2^N), r)$, $r' = r$ where $k \geq 0$ is the first integer such that $(\lfloor\sqrt{r}\rfloor + k)^2 mod(2^N) \leq r$ but $(\lfloor\sqrt{r}\rfloor + k + 1)^2 mod(2^N) > r$.

5. ToyPL denotational semantics

Let us denote by $VARS$ the set of all variables admissible in ToyPL and consider the set of all total functions $ss : VARS \to \mathbb{Z}/2^N$. Let us denote this function space by $SupSts$. The elements of this function space are

"super-states" since they assign values to all variables, while any particular state assigns values to variables of a particular ToyPL-program. Thus, for every super-state ss and every ToyPL-program α, the restriction $(ss \restriction \alpha)$ of ss on the set $VAR(\alpha)$ is a state of α.

Let us consider the space $SupSts \Rightarrow SupSts$ of all *partial* functions $f : SupSts \to SupSts$. Some useful functions in this set are defined below:

- *abort* is the totally undefined function $\emptyset$ on super-states;
- *skip* is the identical function $\lambda ss.ss$ on super-states;
- for every condition ϕ the following *test* function $\phi? = \{(ss, ss) : ss \models_{\mathbb{Z}/2^N} \phi\}$ is a (partial) function on super-states;
- for every variable x and every expression t the following *update* function $UPD_{x,t} = \lambda ss.upd(ss, x, ss(t))$ is a total function on super-states.

At the same time, several *operations* can be defined on functions in $SupSts \Rightarrow SupSts$:

- unary *identify* operation $I = \lambda F.F$;
- binary *composition* operation "$\circ$" such that $(F \circ G)(ss) = G(F(ss))$ for all functions $F, G \in (SupSts \Rightarrow SupSts)$ and every super-state ss; let us remark that this operation is associative;
- (in)finitary *union* operation "$\bigcup$" such that

$$(\bigcup_{m \in M} F_m)(ss) = F_m(ss), , \text{ if } ss \in dom(F_m)$$

  for every (finite or infinite) family of functions $F_m \in (SupSts \Rightarrow SupSts)$, $m \in M$, with disjoint domains, and every super-state ss.

Some other useful operations can be derived as follows:

- for every $m \geq 0$ a unary *m-power* operation "^{m}" such that $F^0 = skip$, $F^1 = F$ and $F^m = \underbrace{F \circ \cdots \circ F}_{m-\text{times}}$ every function $F \in (SupSts \Rightarrow SupSts)$;
- a binary *union* operation "$\cup$" such that $(F_1 \cup F_2) = \bigcup_{m \in \{1,2\}} F_m$ for all functions $F_1, F_2 \in (SupSts \Rightarrow SupSts)$ with disjoint domains; let us remark that this operation is associative and commutative;
- for every condition ϕ, a binary *choice* operation "IF_ϕ" such that $IF_\phi(F, G) = ((\phi? \circ F) \cup ((\neg\phi)? \circ G))$ for all functions $F, G \in (SupSts \Rightarrow SupSts)$ (since functions $(\phi? \circ F)$ and $((\neg\phi)? \circ G)$ have disjoint domains);

- for every condition ϕ, a unary *loop* operation "WH_ϕ" such that $WH_\phi(F) = (\bigcup_{m\geq 0}((\phi?\circ F)^m \circ(\neg\phi)?)$ for every function $F \in (SupSts \Rightarrow SupSts)$ (since functions $(\phi? \circ F)^m \circ (\neg\phi)?$, $m \geq 0$, have disjoint domains).

Now we are ready to define the *denotational semantics* for ToyPL. Let us consider the following algebra $\langle (SupSts \Rightarrow SupSts),\ I,\ \circ,\ UPD,\ IF,\ WH\rangle$, where

- UPD is the set of all update operations $UPD_{x,t}$ for all variables x and expressions t,
- IF is the set of all choice operations IF_ϕ for every condition ϕ,
- WH is the set of all loop operations WH_ϕ for every condition ϕ.

Using these derived functions, we can assign the elements of the algebra to ToyPL-programs and the operations of this algebra to ToyPL-constructs as follows:

1. $[\![x{:=}t]\!] = UPD_{x,t}$ for all variables x and expressions t;
2. $[\![(...)]\!] = I \equiv \lambda F.F$;
3. $[\![;]\!] = \circ \equiv \lambda F, G.(F \circ G)$;
4. $[\![\texttt{if } \phi \texttt{ then...else...}]\!] = IF_\phi \equiv \lambda F, G.IF_\phi(F, G)$ for every condition ϕ;
5. $[\![\texttt{while } \phi \texttt{ do...}]\!] = WH_\phi \equiv \lambda F.WH_\phi(F)$ for every condition ϕ;
6. $[\![\texttt{prog_constr}(\alpha, \beta, \dots)]\!] = [\![\texttt{prog_constr}]\!]([\![\alpha]\!], [\![\beta]\!], \dots)$
   for every $\texttt{prog_constr} \in \{\ ;\ ,\ \texttt{if} - \texttt{then...else...}, \texttt{while} - \texttt{do...}\}$.

For example, let us calculate the denotational semantics of the program SP:

$[\![\texttt{SP}]\!] = [\![\texttt{if z<0 then ... else ...}]\!]([\![\texttt{z:=-1}]\!],$
$[\![\texttt{x:=0; y:=0; while y}\leq\texttt{z do (y:=y+2*x+1; x:=x+1); x:=x-1}]\!]) =$
$= IF_{\texttt{z<0}}(UPD_{\texttt{z},(2^N-1)},$
$[\![\texttt{x:=0; y:=0; while y}\leq\texttt{z do (y:=y+2*x+1; x:=x+1); x:=x-1}]\!]) =$
$= [\![\texttt{x:=0; y:=0; while y}\leq\texttt{z do (y:=y+2*x+1; x:=x+1); x:=x-1}]\!]) =$
$= [\![;]\!]([\![\texttt{x:=0}]\!], [\![\texttt{y:=0}]\!], [\![\texttt{while y}\leq\texttt{z do (y:=y+2*x+1; x:=x+1)}]\!], [\![\texttt{ x:=x-1}]\!])$
$=$
$= [\![\texttt{x:=0}]\!] \circ [\![\texttt{y:=0}]\!] \circ [\![\texttt{while y}\leq\texttt{z do (y:=y+2*x+1; x:=x+1)}]\!] \circ [\![\texttt{x:=x-1}]\!] =$
$= UPD_{\texttt{x},0} \circ UPD_{\texttt{y},0} \circ [\![\texttt{while y}\leq\texttt{z do (y:=y+2*x+1; x:=x+1)}]\!] \circ$
$UPD_{\texttt{x},\texttt{x}-1}$.

Since only three variables x, y and z occur in the program, we can represent any super-state ss by triples of values $(ss(\texttt{x}), ss(\texttt{y}), ss(\texttt{z}))$, each of which is in the range $[0..(2^N - 1)]$. Then we have:

- $UPD_{\mathtt{x},0} \circ UPD_{\mathtt{y},0} =$
$= \{((p,q,r)\ ,\ (0,q,r)) : p,q,r \in [0..(2^N-1)]\} \circ$
$\circ\ \{((p,q,r)\ ,\ (p,0,r)) : p,q,r \in [0..(2^N-1)]\} =$
$= \{((p,q,r)\ ,\ (0,0,r)) : p,q,r \in [0..(2^N-1)]\};$

- $[\![\mathtt{while\ y\leq z\ do\ (y:=y+2*x+1;\ x:=x+1)}]\!] =$
$= \bigcup_{m\geq 0}(((\mathtt{y} \leq \mathtt{z})? \circ [\![\mathtt{y} := \mathtt{y}+2*\mathtt{x}+1; \mathtt{x} := \mathtt{x}+1]\!])^m \circ (\mathtt{y} > \mathtt{z}?)) =$
$= (\bigcup_{m\geq 0}(((\mathtt{y} \leq \mathtt{z})? \circ [\![\mathtt{y} := \mathtt{y}+2*\mathtt{x}+1; \mathtt{x} := \mathtt{x}+1]\!])^m)) \circ (\mathtt{y} > \mathtt{z}?) =$
$=^6 (\bigcup_{m\geq 0}\{((p,q,r)\ ,\ ((p+m)mod(2^N), (q-p^2+(p+m)^2)mod(2^N), r)) :$
$p,q,r,\in [0..(2^N-1)],\ m \geq 0$, and
$q, (q-p^2+(p+1)^2)mod(2^N), \dots (q-p^2+(p+m-1)^2)mod(2^N) \leq r\}) \circ$
$\circ(\mathtt{y} > \mathtt{z}?) =$
$= \{((p,q,r)\ ,\ ((p+m)mod(2^N), (q-p^2+(p+m)^2)mod(2^N), r)) :$
$p,q,r,\in [0..(2^N-1)],\ m \geq 0$,
and $q, (q-p^2+(p+1)^2)mod(2^N), \dots (q-p^2+(p+m-1)^2)mod(2^N) \leq$
r, but $(q-p^2+(p+m-1)^2)mod(2^N) > r\};$

- $UPD_{\mathtt{x},\mathtt{x-1}} = \{((p,q,r)\ ,\ ((p-1)mod(2^N), q, r)) : p,q,r,\in [0..(2^N-1)]\}.$

Combining together, we conclude that the denotational semantics of the sample ToyPL-program SP is the following partial function $[\![\mathtt{SP}]\!] : SupSTS \to SupSts$, which maps a super-state (p,q,r), $p,q,r,\in [0..(2^N-1)]$, into another super-state $((m-1)mod(2^N), (m^2)mod(2^N), r)$, where $m \geq 0$, and $1^2 mod(2^N) \leq r$, ... $(m-1)^2 mod(2^N) \leq r$, but $(m^2)mod(2^N) > r$.

Structural operational semantics is sound and complete with respect to denotational semantics for ToyPL in the following sense.

Statement 6. *For every ToyPL-program α the following holds:*

Soundness: *For all states s', s'', if $\vdash_{ToyPL} s'\langle\alpha\rangle s''$ then $[\![\alpha]\!](ss') = ss''$ for all super-states $ss' = (ss' \upharpoonright \alpha)$ and $ss'' = (ss'' \upharpoonright \alpha)$ that are equal on all variables in $VARS \setminus VAR(\alpha)$.*

Completeness: *For all super-states ss' and ss'', if $[\![\alpha]\!](ss') = ss''$, then $\vdash_{ToyPL} (ss' \upharpoonright \alpha)\langle\alpha\rangle(ss'' \upharpoonright \alpha)$.*

Proof hint: soundness — induction on the height of the inference tree, completeness — induction on the program structure. Please see for details the textbook *Introduction to Parsing, Semantics, Compilation and Verification of Programs.* ■

[6] Use induction by $m \geq 0$. Details are available in the textbook *Introduction to Parsing, Semantics, Compilation and Verification of Programs.*

6. Conclusion

Here are, once again, the basic theses of the paper.

1. There is a demand for popular Formal Semantics due to (at least) its educational value: it helps to discipline minds. Yet, contemporary educational environment (in Computer Science) is reluctant to teaching/learning Formal Semantics since it is too pure in theory and poor in practice.
2. A major obstacle to popularizing Formal Methods, and Formal Semantics in particular, is the absence of elementary explanatory examples for primary teaching/learning. For this reason, we have presented in this paper several examples of *toy* formal semantics used for teaching students at the Department of Mechanics and Mathematics and Information Technology Department of Novosibirsk State University.
3. As soon as we turn from elementary examples (like ToyPL) to practical programming languages, the formal semantics becomes much more complicated. The situation becomes extremely intricate (and, maybe, inadmissible) if we attempt to develop a comprehensive semantics for programming languages with pointers, memory allocation and release, and objects.

The best opportunity to overcome the complexity/feasibility problem for comprehensive formal semantics of programming languages may be a switch to problem-oriented semantics. For example, in the recent paper *Steps Towards a Theory and Calculus of Aliasing* (to appear in International Journal of Software and Informatics, 2011), Bertrand Meyer suggested a problem-oriented denotational semantics that he called *Calculus of Aliasing*. This semantics is designed to detect defects/mistakes/problems when several different "expressions" point to the same memory location(s) simultaneously. Axiomatic semantics for the same purpose is known as *Separation Logic*. It was suggested by John C. Reynolds a decade ago in the paper *A Logic for Shared Mutable Data Structures* (IEEE Symposium on Logic In Computer Science, 2002). Relation between these two formalisms is a topic for further research.

References

[1] Dijkstra E.W. On a cultural gap // The Mathematical Intelligencer. – 1986. – Vol. 8, N 1. – P. 48–52.

[2] Floyd R.W. Assigning Meanings to Programs // Proc. Symp. Applied Mathematics, Am. Mathematical Soc. – 1967. – Vol.19. – P. 19–31.