

Using Esoteric Language for Teaching Formal Semantics

N. V. Shilov

School of Science and Technologies, Nazarbayev University**
nikolay.shilov@nu.edu.kz

Abstract. The paper presents an approach to undergraduate teaching of formal semantics of programs. It explains what is formal semantics by developing operational, denotational and axiomatic semantics for an “esoteric” language. We hope that the approach will be interesting to communities of Program Theoreticians and Software Engineers, that it will help better education to bridge a cultural gap between Formal Methods theories and Software Engineering practice.

1 Introduction

45 years have passed since Robert W. Floyd published the first research that explicitly discussed formally how to “assign meaning to programs”, i.e. formal program 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, v.32). These four decades (as well as the last one) have generated a number of individual formal semantics, their theories, and *more experts, but fewer general users* (D.A. Schmidt). Meanwhile, we could see a great activity (of different kinds) in the field of formal semantics in the last decade.

But recently David L. Parnas 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, v.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 (Mikheylo) Vasilyevich Lomonosov reads¹: *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. The reason to quote the above aphorism here is related to educational value of Mathematics: it disciplines minds for a general education. We do believe that Formal

** While on leave from A. P. Ershov Institute of Informatics Systems, Novosibirsk, Russia

¹ translated by the authors

Methods (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 opposite, we must care!). But 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 arithmetic from Peano axiomatic, but it is a common sense to start from elementary problems about numbers of apples, pencils, etc. For example, nobody teaches primary school children 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 then he gave 2 apples to John. How many apples does Peter have after that?

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

In our vision, a part of the reason of student's and engineer's poor demand in Formal Methods, is very simple: FM-experts do not care about primary education in this field at the 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 a one-semester course for the first-year students *Programming-I (Introduction to Computer Science and Programming Languages)* at the Mathematics and Mechanics Department of the Novosibirsk State University in 2008-2010.

Acknowledgement: This work is part of the project *Research of formal Models for analysis of programs with Dynamic Memory* funded by Nazarbayev University Research and Innovation System.

Disclaimer: This paper is not a research in Computer Science or Software Engineering in *pure* sense, but a sketch of an approach to undergraduate 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 formal reference list includes real classics of the past only, but no references to contemporary papers, pointers to contemporary research papers are inlined. This teaching experience was reported at a workshop on Program Semantics, Specification and Verification PSSV-2012 (July 1-2, 2012, Nizhni Novgorod, Russia) affiliated with 7th International Symposium Computer Science in Russia; the report has been published in the proceedings of the workshop.

```

⟨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⟩

```

Fig. 1. BNF definition of the TEL syntax

2 A Toy Esoteric Language TEL

A programming language is any artificial computer language that is designed for 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 use of the meaningful syntactically-correct constructs.

2.1 Formal Syntax, Informal Semantics, and Pragmatics of TEL

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

Definition of the TEL syntax is given in Figure 1. 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 TEL program is given in Figure 2, let us refer to this program as the sample program **SP** throughout the paper.

Semantics of the language is non-conventional, non-programming, just artificial. It can be defined informally as follows: semantics of a TEL 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 TEL 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 TEL (in spite of non-conventional non-programming semantics).

```

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)

```

Fig. 2. A sample TEL program SP

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

2.2 Ways to Assign Meanings to TEL

Operational Semantics: executable state machines Operational semantics *translates* programs to corresponding *state machines* (“mechanical procedures” of a certain class) whose “operations” (that change machine’s state) are (conventionally) “executable”: semantics of a program is defined in terms and by means of all admissible “executions” (i.e. runs) of the corresponding machine.

In the case of TEL, the target class of machinery consists of recursive algorithms (without input) that compute integer values. A recursive algorithm that corresponds to a program α is denoted by F_α ; the operational semantics of α is the value that is computed by F_α , it is denoted by F_α also. The translation (the correspondence) is defined as follows:

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

As follows from the definition, the recursion proceeds by the syntax tree. For every program α the states of the machine F_α are all evaluation steps of F_α . In particular, the operational semantics of the sample program SP is 1 since the following computation is a run of the algorithm F_{SP} :

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

In particular, states of the algorithm (machine) F_{SP} are all lines of the above run.

Let us remark that as soon as a formal semantics is provided for a language, it induces an equivalence relation on programs. For example, TEL 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.

Denotational Semantics: an algebra for calculations An *algebra* is a set of objects with operations on them. For example, natural numbers $\mathbb{N}$ 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 $\mathbb{N}$ 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 $\llbracket \cdot \rrbracket$.

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

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

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

$$\begin{aligned}
& \llbracket \text{if } z < 0 \text{ then } z := -1 \\
& \quad \text{else } (x := 0 ; y := 0 ; \\
& \quad \quad \text{while } y \leq z \text{ do } (y := y + 2 * x + 1 ; x := x + 1) ; \\
& \quad \quad x := x - 1) \rrbracket = \\
& = \llbracket \text{if-then...else...} \rrbracket \\
& \quad (\llbracket z := -1 \rrbracket, \llbracket x := 0 ; y := 0 ; \\
& \quad \quad \text{while } y \leq z \text{ do } (y := y + 2 * x + 1 ; x := x + 1) ; \\
& \quad \quad x := x - 1 \rrbracket) = \\
& = \min\{ 1, \llbracket ; \rrbracket(\llbracket x := 0 \rrbracket, \llbracket y := 0 ; \\
& \quad \quad \text{while } y \leq z \text{ do } (y := y + 2 * x + 1 ; x := x + 1) ; \\
& \quad \quad x := x - 1 \rrbracket) \} = \\
& = \min\{ 1, 1 + \llbracket ; \rrbracket(\llbracket y := 0 \rrbracket, \\
& \quad \quad \llbracket \text{while } y \leq z \text{ do } (y := y + 2 * x + 1 ; x := x + 1) ; \\
& \quad \quad x := x - 1 \rrbracket) \} = \\
& = \min\{ 1, 1 + (1 + \dots) \} = 1.
\end{aligned}$$

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 TEL are closely related.

Statement 1 $F_\alpha = \llbracket \alpha \rrbracket$ for every TEL program α .

Proof hint: induction on the program structure. ■

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

Table 1. TEL axiomatic semantics

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

Axiomatic Semantics: code-driven proofs 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 TEL is an axiomatic system for *assertions* of the following form $\langle \text{constant} \rangle \leq \langle \text{program} \rangle \leq \langle \text{constant} \rangle$, where each constant is a non-negative integer or infinity ∞ , but the former should not be greater than the latter.

TEL axiomatic semantics comprises the axioms and inference rules presented in Table 1.

An assertion $m \leq \alpha \leq n$ is said to be valid, if $m \leq \llbracket \alpha \rrbracket \leq n$ (or, according to soundness of TEL 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 *TEL axiomatic semantics is sound and complete.*

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

3 Concluding Remarks

There is a demand in popular formal program semantics due to (at least) its educational value: it helps to discipline minds. One of the major obstacles for

popularizing formal program semantics is the absence of elementary explanation examples for primary teaching/learning. Due to this reason we presented in this paper several examples of *esoteric* formal semantics that are used for teaching students at the Mathematics & Mechanics and Information Technology Departments of the Novosibirsk State University. Unfortunately, as soon as we turn from esoteric and toy examples to practical programming languages, the formal semantics becomes much more complicated. The situation becomes extremely complicated (and, maybe, non-feasible), if we attempt to develop a comprehensive semantics for a programming language with pointers, memory allocation and release, and objects.

Maybe the best opportunity to overcome the complexity/feasibility problem for comprehensive formal semantics of programming languages is a switch to problem-oriented semantics. For example, in a 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.

Acknowledgement: The author would like to thank the anonymous reviewer for many corrections.

References

1. Dijkstra E.W. *On a cultural gap*, The Mathematical Intelligencer **8(1)** (1986), 48–52.
2. Floyd R.W. *Assigning Meanings to Programs*, Proc. Symp. Applied Mathematics, Am. Mathematical Soc. **19** (1967), 19–31.