

# Microsemantics as a Bootstrap in Teaching Formal Methods

Raymond Boute  
INTEC — Universiteit Gent, Belgium  
*Raymond.Boute@intec.UGent.be*

## Abstract

**Introducing an elementary form of program semantics early in the curriculum provides a good preamble to formal methods. Microsemantics uses only the most basic concept in formal mathematics, namely *substitution*, and therefore can be presented as early as the second lecture of a freshman-level course. It can subsequently serve as a bootstrap for gradually introducing most of the other fundamental concepts of formal methods, leading up to formal systems specification and design.**

*Keywords: calculational reasoning, domain-independent problems, formal methods, lambda calculus, microsemantics, state equations, substitution, program semantics, teaching*

## 1. INTRODUCTION

In a recent paper [7], we argued that the best way for making formal methods widely accepted as a self-evident part of professional practice in computer-related areas of engineering is by ensuring that our educational institutions provide a massive injection of qualified people into industry.

Doing so requires a curriculum where formal methods are not taught in a few isolated courses, but rather where *all* courses set the example by systematically using mathematical modeling — which, as Parnas [15] aptly observes, is the major ability distinguishing professional engineers from other designers. This view underlies the common educational model in classical engineering, but many computer-related curricula and courses have considerable catching up to do.

Solid foundations upon which “all” other courses can rely clearly must be laid at the start of the curriculum. In order to make the inherently high abstraction level accessible to most students, it is necessary to provide a variety of conceptual and application illustrations and exercises. Yet, in the first year most students still have very limited domain-specific (technical) knowledge. For this and other reasons, it is advantageous to use domain-independent examples and problems [7].

The supply of such problems is very rich and varied. However, the solutions typically given are meant to show cleverness rather than systematic formulation and reasoning. Often not only the reasoning steps but even the initial casting into mathematical formulas is coarse and has serious gaps. Filling those gaps, in both expression and reasoning, constitutes an added dimension.

In the same paper [7], we also considered a class of domain-independent problems or “puzzles” where the gaps are due to the fact that the problem is stated procedurally or operationally, whereas the solution must be succinct and declarative. Here is an example of such a problem, found on a list of problems posted by Dahlke [8], but reformulated here for the sake of brevity.

A school has 1000 students and an array of 1000 lockers, all initially closed. All students walk along the row one after the other, and the  $k^{\text{th}}$  student inverts the state of every  $k^{\text{th}}$  locker, that is: opens the locker if it was closed and vice versa. How many lockers are open in the end?

Since classical mathematics has no framework for describing procedures, traditional solutions make a fairly large jump from the informal procedural statement to the mathematical equations from which the final solution is derived. Programming languages on the other hand are suitable for expressing procedures, and hence can help in accurately formalizing the procedure expressed in words in the problem statement. Here is such a formalization for the preceding example:

for  $k$  in  $1..K$  do (for  $n$  in  $1..N$  do if ( $k$  divides  $n$ ) then inv ( $L$   $n$ ) fi od) od ,  
 assuming  $K$  students and an array  $L$  of  $N$  lockers (defining  $inv$  for inversion is not relevant here).

Yet, using a programming language reduces only the gap between informal and formal problem statements at the procedural level, but the gap between the procedure and the mathematical equations leading to the final solution remains and has become even more visible.

Clearly, the bridge from program to equations is provided by mathematical semantics. More specifically, we need a formalism to derive, from a program text, mathematical equations that express the program's behavior. This amounts to mathematically modeling a procedure in the same spirit as an electronics engineer models a circuit by deriving, from the circuit diagram, mathematical equations that express the circuit's behaviour.

To be suitable already within the first weeks of a freshman-level course, the formalism should be very lightweight, and in proportion to the perceived simplicity of the puzzles of interest. Our approach, first outlined in [7], meets this requirement, and is therefore named *microsemantics*.

The principles of microsemantics, guidelines for its use and a few examples are presented in section 2, whereas section 3 outlines how it can be used for bootstrapping fundamental concepts such as equational reasoning, lambda calculus, propositional and predicate logic, finally leading to formal systems specification and design at the end of such a course.

## 2. MICROSEMANTICS

For achieving the extremely lightweight formalism needed for the stated purpose, we use only the most basic concept in formal mathematics, namely *substitution*, as described, for instance, by Gries and Schneider in [10, Chapter 1]. This is the limited toolkit assumed at this stage.

**Substitution** Substituting expression  $d$  for variable  $v$  in expression  $e$  is written  $e[e^v_d]$ . An example:

$$(x + 3 \cdot y)_{z+1}^y \stackrel{s}{=} x + 3 \cdot (z + 1) .$$

We use  $\stackrel{s}{=}$  for syntactic equality. A warning: if parentheses in  $d$  are omitted according to the usual conventions, it may be necessary to reinstate them for substitution, e.g., around  $z + 1$  in the preceding example. By an obvious generalization,  $v$  and  $d$  may be tuples, as illustrated by

$$(x + 3 \cdot y)_{a \cdot y, z+1}^{x, y} \stackrel{s}{=} a \cdot y + 3 \cdot (z + 1) .$$

Multiple (“simultaneous”) substitutions do not necessarily have the same as successive substitutions. With the general convention that one may write  $e[e^v_c]$  for  $(e[e^v_c])^w_d$ , this is shown by

$$(x + 3 \cdot y)_{a \cdot y}^{x, y} [z+1]^y \stackrel{s}{=} (a \cdot y + 3 \cdot y)_{z+1}^y \stackrel{s}{=} a \cdot (z + 1) + 3 \cdot (z + 1) .$$

One can see  $[^v_d]$  as a (metalevel) syntactic operator on mathematical expressions.

**Program equations** The *state* is a tuple whose elements correspond to the program variables, in the order and with types as declared. Let  $s$  be the tuple made from the variables themselves. Such a *variable list* turns out to be useful in unifying theories [6] and in microsemantics.

The term *command* designates instructions as well as entire programs. A command  $c$  is modeled as a function from states to states, defined by an axiom of the form  $cs = e$ . This is a familiar style for defining functions in mathematics, as shown in defining square by square  $x = x \cdot x$ . Such axioms are “used” or “applied” by *instantiation* [10, Chapter 1]; more specifically,  $(cs = e)_{a}^s \stackrel{s}{=} cd = e_{a}^s$ . For substitution, commands are treated as function constants [10, Chapter 1], i.e., are not affected.

First we consider the *primitive* commands *assignment* ( $v := e$ ), *composition* ( $c ; c'$ ), *selection* (if  $b$  then  $c$  else  $c'$  fi), where  $c$  and  $c'$  are commands and  $b$  a boolean expression:

$$(v := e) s = s[e^v] \tag{1}$$

$$(c ; c') s = c'(cs) \tag{2}$$

$$(\text{if } b \text{ then } c \text{ else } c' \text{ fi}) s = b ? cs \uparrow c's . \tag{3}$$

The last right hand side is a *conditional expression*. Its general form is  $b ? e_1 \dagger e_0$ , read in words as “if  $b$  then  $e_1$  else  $e_0$ ” and formalized by the axiom  $(b ? e_1 \dagger e_0) = e_b$  where  $b$  is 0 or 1. Using  $s$  as the formal argument in (1) and (3) ensures their simple and intuitive form; for (2) it is unimportant.

The following are *derived* commands, since they can be defined in terms of the primitive ones.

$$\text{skip} = (v := v) \quad (\text{or, as a primitive command, } \text{skip } s = s) \quad (4)$$

$$(\text{if } b \text{ then } c \text{ fi}) = (\text{if } b \text{ then } c \text{ else skip fi}) \quad (5)$$

$$(\text{while } b \text{ do } c \text{ od}) = (\text{if } b \text{ then } (c ; \text{while } b \text{ do } c \text{ od}) \text{ fi}) . \quad (6)$$

Observe that microsemantics is fully minimalist in the sense that it involves no other notation than the basic expressions and commands that may occur in the simple programs themselves and substitution. The machinery of denotational and axiomatic semantics, which may be intimidating if presented in the first few weeks, is bypassed. Of course, it can be introduced at a later stage.

**Guidelines for use and examples** As a matter of convention, we write program variables in typewriter font (e.g.,  $x$ ) and the matching state variable in italic (e.g.,  $x$ ). This distinction has no calculational consequences and is for exposition only: if it helps, fine, otherwise it can be safely ignored (which is convenient in handwriting, where the distinction is easily lost anyway).

*About assignment* We start with an example. Assume a program with variables  $x$  and  $y$ , declared as integers, so the state  $s$  is the pair  $(x, y)$ . For the assignment  $x := x + y$ ,

$$\begin{aligned} (x := x + y) (x, y) &= \langle \text{Axiom (1)} \rangle (x, y)_{x+y}^x \\ &= \langle \text{Substitut.} \rangle (x + y, y) . \end{aligned} \quad (7)$$

Axiom (1) also covers multiple assignment, i.e., of the form  $(v := e)$  where  $v$  is a tuple of variables and  $e$  is a tuple of expressions. An example is swapping two variables:

$$\begin{aligned} (x, y := x, y) (x, y) &= \langle \text{Axiom (1)} \rangle (x, y)_{y,x}^{x,y} \\ &= \langle \text{Substitut.} \rangle (y, x) . \end{aligned}$$

*Instantiating correctly* Clearly  $(x := x + y) (2, 3) = (5, 3)$  by instantiating the result from (7). More generally, recall that instantiating  $cs = e$  by  $[d^s]$  yields  $cd = e[d^s]$  and hence, for Axiom (1),

$$(v := e) d = (s_{[e]}^v)_{[d]}^s . \quad (8)$$

Warning: it is tempting but wrong to write  $(s_{[e]}^v)_{[d]}^s$  as  $d_{[e]}^v$ . Interestingly, this warning is more relevant for advanced users prone to skip steps than for beginners who tend to apply the rules meticulously.

*About program structure* Observe the style of (1..6), where  $c$  and  $c'$  in turn stand for arbitrary commands. By saying that the left hand sides (omitting  $s$ ) are the only forms for commands, the syntax is specified as well. More specifically, the structure of a program is described recursively in the sense that it is either the most elementary command (i.e., an assignment), or a composition of two commands, or a selection between two commands. The latter commands are again arbitrary: each of them can again be an assignment, a composition or a selection etc.. Axioms (1..6) illustrate how a (semantic) function can be defined by recursion on (program) structure.

*About composition* At this stage, we can prove an interesting algebraic program property:

$$(c ; (c' ; c'')) s = ((c ; c') ; c'') s \quad (\text{exercise}) . \quad (9)$$

This means that  $(c ; (c' ; c'')) = ((c ; c') ; c'')$ , at least *for the particular state semantics considered in this note*. As is customary with associative operators in mathematics, we make the parentheses in composition optional. We also make other parentheses optional insofar as replacing them is “obvious” (in a formalizable way). This gives rise to an interesting illustration about handling syntactic and semantic ambiguity, discussed elsewhere in the course.

The typical error to which we alluded in the warning about instantiating correctly is shown in the sloppy calculation  $(c ; v := e) s = (v := e) (cs) = (cs)_{[e]}^v$ , whereas (8) yields the correct  $(s_{[e]}^v)_{[cs]}^s$ .

It is instructive to exercise with successive assignment. An example is swapping  $x$  and  $y$  using an auxiliary variable  $z$ , as done by the program  $z := x ; x := y ; y := z$ . Here is a fully detailed calculation for the final state if the initial state is  $x, y, z$  (exercise: using  $a, b, c$  instead).

$$\begin{aligned}
 (z := x ; x := y ; y := z) (x, y, z) &= \langle \text{Ax. (2)} \rangle (y := z) ((x := y) ((z := x) (x, y, z))) \\
 &= \langle \text{Ax. (1)} \rangle (y := z) ((x := y) (x, y, z) \overset{z}{x}) \\
 &= \langle \text{Subst.} \rangle (y := z) ((x := y) (x, y, x)) \\
 &= \langle \text{Ax. (1)} \rangle (y := z) (x, y, z) \overset{x}{y} \overset{z}{x, y, x} \\
 &= \langle \text{Subst.} \rangle (y := z) (y, y, z) \overset{x, y, z}{x, y, x} \\
 &= \langle \text{Subst.} \rangle (y := z) (y, y, x) \\
 &= \langle \text{Ax. (1)} \rangle (x, y, z) \overset{y}{z} \overset{x, y, z}{y, y, x} \\
 &= \langle \text{Subst.} \rangle (x, z, z) \overset{x, y, z}{y, y, x} \\
 &= \langle \text{Subst.} \rangle (y, x, x) .
 \end{aligned}$$

This shows both the power of substitution (it can do the job all by itself) and its limitations (tedious).

*Selection and iteration* Familiarizing the students with selection is fairly straightforward. The equations for loops are recursive, and therefore solving them is a good introduction to recursion and iteration. This is a topic for later in the course, as outlined next.

### 3. MICROSEMANTICS FOR BOOTSTRAPPING FUNDAMENTAL CONCEPTS

Microsemantics provides a running example for gradually introducing the fundamental concepts underlying formal methods in an introductory course. The following is just an outline.

**Syntax, axiomatics, semantics, pragmatics** Generally it is not a good idea to burden beginning students with technicalities. However, microsemantics offers opportunities for giving a simple and concrete first explanation about the distinctions and relations between syntax (here: expressions and commands), axiomatics (formal calculation rules), semantics (various views on meaning) and pragmatics (to what purpose notation and rules are designed in a particular way).

**Equational reasoning** The earlier calculation already provides an example of equational reasoning, although it involves a purely syntactic (meta-)operator, namely,  $-\square$  for substitution. The obvious next step is equational reasoning involving only functions and expressions in [10]. The language (basic mathematics), for which it suffices here to mention the the first chapter in [10].

**Lambda calculus** Lambda notation is intuitive enough to be understood as a “finished product”. However, after microsemantics one can provide an extra dimension, showing how to “re-invent” lambda calculus from the pragmatics (usage) viewpoint by making optimal design decisions. We start by analogy with the convention of *partial application* [5] of mathematical operators. This means that the application of an operator to part of its argument forms an operator that can be applied to the rest<sup>1</sup>. Using this convention for the meta-operator for substitution makes the earlier  $[\overset{v}{d}]$  one of the possible partial applications. However,  $[\overset{v}{d}]$  has  $v$  as a “dangling reference”, which permits only inelegant calculation rules. A different partial application that has no dangling references<sup>2</sup> and hence permits elegant calculation rules is  $e^{[v]}$ . For this, we adopt the alternative notation  $\lambda v.e$ , which is the basic form for (impure) lambda terms. Clearly,  $(\lambda v.e)d = e^{[\overset{v}{d}]}$ . Now we can also rewrite axioms of the form  $cs = e$  as  $c = \lambda s.e$ . For instance, axiom (1) can be written

$$(v := e) = \lambda s.(\lambda v.s)e .$$

Similarly, lambda combinators can be designed for composition and selection, which readers familiar with lambda calculus will see as straightforward. This sets the stage for presenting formal calculation rules for lambda terms [2] followed by a first introduction to variable-free expression with combinatory logic [2] as an archetype, and generic functionals [4] for practical use.

<sup>1</sup>This must not be confused with currying: it pertains to an operator taking a tuple as its only argument, and a partial argument then consists of some elements of the tuple.

<sup>2</sup>For the “insiders”: not to be confused with free variables; these may still be present.

**Propositional and predicate logic** As soon as programs involve a few conditionals and domains of discourse other than basic arithmetic, the need for propositional and predicate logic arises, and can be satisfied by the proper chapters in [10] or by our own functional variant [3, 5].

**Induction** As soon as programs with loops are considered, the equations describing their behavior are recursive, and calculational reasoning about this is best supported by induction. We have found that notions about induction that students may have retained from previous courses, e.g., in high school, may lead them into all kinds of pitfalls. Most typical errors amount to instantiating the induction hypothesis. So here is the opportunity for setting things straight by explaining the structure of inductive proofs, the relation with well-foundedness, various proof styles, other forms of induction than over numbers only (structural, e.g., over lists) and so on.

**Relations** In the form presented, microsemantics uses ante/post functions, which covers only deterministic programs. By analogy with certain state equations in elementary mechanics [6], we introduce *ante-expressions*  $\backslash e$  derived from expressions  $e$  by replacing all state variables  $v$  by  $\backslash v$  (using *ante-quotes*), corresponding to the state before a command is executed. Note that, by this convention,  $\backslash e = e[\backslash s]$ . Similarly,  $e' = e[s']$  for *post-expressions* corresponding to the state after a command is executed. The functional model of the form  $cs = e$  (for a command  $c$ ) can thereby be generalized to a relational model of the form<sup>3</sup>  $s cs' \equiv p$  as Hehner's book [11] if only one kind of semantics is wanted, or  $\backslash s R_c s' \equiv p$  as in [6] if several semantics are wanted. Hence relations pave the way to nondeterministic programs and formal specifications. Furthermore, at this stage of the course, a sufficient basis is available for discussing extremal elements (maximum and minimum, greatest and least, upper and lower bounds), the various kinds of orderings (preorder, partial order etc.) most encountered in the theory and practice of programming, and the various styles of calculating with relations (point-wise, point-free).

#### 4. RELATION TO OTHER COURSES AND TOPICS

**Formal specification and design of systems** For a first-year course, the preceding material will usually need spreading over a complete semester (say, 2 lectures and 2 exercise sessions<sup>4</sup> per week during 12 weeks) for allowing the students to develop adequate understanding and skills. Yet, it would benefit from further consolidation and practicing by a second semester on formal systems specification and design. The emphasis can be on systems theory and hybrid systems (in the spirit outlined by Lee and Varaiya [14]) or on software systems (program semantics, model checking etc. [13]) depending on whether the course is meant to be EE or CS oriented.

**Courses on programming** Optimally, a course as outlined in Section 3 should start no later than introductory programming, allowing the latter to be taught with proper foundations and attitudes. Moreover, for introductory programming, functional languages would be most suitable, although few universities seem to have the courage to make such a decision. Its functional style makes microsemantics a convenient early link between imperative and functional languages. However, it is meant only as a bootstrap assuming minimal prerequisites, whereas, for realistic program derivation, practical methods in the line of axiomatic semantics must be covered [6, 11].

**Courses on formal semantics** Formal semantics is to the software engineer what device and circuit theory is to the electronics engineer. Its elimination from certain curricula (in favor of basic courses on programming languages that any engineering or CS student should be able to learn independently as part of the preparation for the exercise sessions of other courses) is unjustified. However, a course as outlined in Section 3 would lower the threshold for (re-)introducing formal semantics in the spirit of unifying theories of programming [6, 12]

Taking microsemantics as an introductory example, its unification with axiomatic semantics was outlined earlier in the context of relations. Here we outline its unification with the classical denotational style. Let the denotational-style meaning functions be  $\mathcal{C}$  for commands and  $\mathcal{E}$  for expressions. For instance, for single-variable assignment,  $\mathcal{C}(v := e) \sigma = (v \mapsto \mathcal{E} e \sigma) \otimes \sigma$  where,

<sup>3</sup>Aside: Hehner does not use antequotes, and just writes  $s$ . In [6] we show when and how antequotes are convenient.

<sup>4</sup>Assuming 75 minutes each, the standard at Ghent University. Arguably, for lectures  $3 \times 50$  min. is better than  $2 \times 75$  min.

as usual (up to notational differences), the state  $\sigma$  is a function from variables to denotations,  $\mapsto$  makes maplets, and  $\otimes$  is function overriding. Since the interpretation of mathematical equations is common to both semantics and hence trivial for the unification, we factorize it out and take the common denotation domain to be the set of expressions, keeping everything purely syntactic. Writing  $\mu$  for the mapping from denotational-style states to microsemantics-style states, unification is established by proving  $\mu(\mathcal{C}c\sigma) = \mathcal{M}c(\mu\sigma)$  for any command  $c$  and denotational-style state  $\sigma$ , where we made the microsemantics denotation function  $\mathcal{M}$  explicit just to eliminate its hitherto privileged status. For instance, for assignment we must prove  $\mu((v \mapsto \mathcal{E}e\sigma) \otimes \sigma) = (s[e^v]_{\mu\sigma}^s)$ . Using the properties of generic functionals [4], this takes only a few lines.

## 5. CONCLUSIONS

We have outlined how a first-year preamble to formal methods can be constructed around microsemantics. Of course, for the sake of diversity, other kinds of application examples and problems must be included as well, preferably domain-independent [7] for the majority, but also some domain-specific cases requiring only prior knowledge from high school or parallel courses.

Our own courses of this kind [3] emphasize clean formalization and calculational reasoning [10] right from the beginning, placing the insights gained from Computing Science [9] at the disposal of our students. Avoiding poor practices is always easier than correcting them afterwards.

A final practical remark: microsemantics is too recent for having been incorporated in [3], but will appear in a future version. We hope that, meanwhile, other teachers may benefit from the idea.

## REFERENCES

- [1] Roland Backhouse (2005), *Algorithmic Problem Solving*. Lecture Notes, University of Nottingham. On the web: <http://www.cs.nott.ac.uk/~rcb/G5AAPS/aps.ps>
- [2] Henk P. Barendregt (1984), *The Lambda Calculus, Its Syntax and Semantics*, North-Holland.
- [3] Raymond Boute (2002), *Functional Mathematics: a Unifying Declarative and Calculational Approach to Systems, Circuits and Programs — Part I*. Course notes, Ghent University. On the web via [www.funmath.be](http://www.funmath.be)
- [4] Raymond Boute (2003), Concrete Generic Functionals: Principles, Design and Applications, in: Jeremy Gibbons and Johan Jeuring, eds., *Generic Programming*, 89–119, Kluwer.
- [5] Raymond Boute (2005), Functional declarative language design and predicate calculus: a practical approach, *ACM Trans. Prog. Lang. Syst.*, **27**, 988–1047.
- [6] Raymond Boute (2006), Calculational semantics: deriving programming theories from equations by functional predicate calculus, *ACM Trans. Prog. Lang. Syst.*, **28**, 747–793.
- [7] Raymond Boute (2006), Using Domain-Independent Problems for Introducing Formal Methods, in: Jayadev Misra, Tobias Nipkow, Emil Sekerinski, eds., *FM 2006: Formal Methods*, 316–331. Springer LNCS 4085.
- [8] Karl Dahlke, *Fun and Challenging Math Problems for the Young, and Young At Heart*, on the web: <http://www.eklhad.net/funmath.html>
- [9] Edsger W. Dijkstra (1990), How Computing Science created a new mathematical style, *EWD 1073*. University of Texas at Austin. On the web: <http://www.cs.utexas.edu/users/EWD/ewd10xx/EWD1073.pdf>
- [10] David Gries and Fred Schneider (1993), *A Logical Approach to Discrete Math*. Springer.
- [11] Eric C. R. Hehner (2004), *A Practical Theory of Programming* (2nd edition). Springer. Regularly updated version on the web: <http://www.cs.toronto.edu/~hehner/aPToP>
- [12] C. Antony R. Hoare, He Jifeng (1998), *Unifying Theories of Programming*. Prentice-Hall.
- [13] Leslie Lamport (2002), *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley.
- [14] Edward A. Lee and Pravin Varaiya (2000), Introducing Signals and Systems — The Berkeley Approach. *First Signal Processing Education Workshop*, Hunt, Texas, Oct. 15–18. On the web: <http://ptolemy.eecs.berkeley.edu/publications/papers/00/spe1>
- [15] David L. Parnas (1993), Predicate Logic for Software Engineering, *IEEE Trans. SWE*, **19**, 856–862.