Algebraic Operational Semantics and Modula-2*

Yuri Gurevich and James M. Morris
Electrical Engineering and Computer Science
University of Michigan
Ann Arbor, MI 48109-2122

0. Introduction

We start with several arguments in favor of operational semantics for imperative programming languages. One important purpose of formal semantics is to help a programmer understand a given language (as opposed to particular programs written in that language). We would claim that, when conceiving a program expressed in an imperative language, a journeyman programmer has in mind an ideal machine that executes the language's commands. That is to say, our fundamental understanding of an imperative programming language is behavioral (or operational).

A semantic description of a programming language should provide an accessible account of all of a language's constructs. Languages like Modula-2 that are designed for (among other things) the writing of operating systems include facilities for multiprocessing and facilities for describing interaction with peripheral devices. Specifically, such languages include a means for dealing with hardware interrupts which usually involve both these sorts of facility. Consequently, an adequate semantic account of a language like Modula must treat interrupts. Now, the very notion of interrupt involves the concept of time: an interrupt is an event which occurs at an arbitrary moment in a computation. The idea that a computation is a sequence of states unfolding in time is the basis of operational semantics. Therefore, it seems most natural to describe operationally languages which allow one to deal with interrupts. It also seems to be true that programming language constructs for expressing multiprocessing are most straightforwardly described in terms of the behavior they elicit.

A formal semantics for a language should also provide a basis for proving the correctness of the language's implementations, for examining the expressive power of the language, for reasoning about programs written in the language, etc. We emphasize that operational semantics provides only a basis rather than methods for accomplishing these tasks. Some of the tasks fall within the purview of a logic or proof system using operational semantics as a foundation. We no not consider operational semantics as a competitor of other approaches, like axiomatic or denotational semantics or temporal logic, but rather as complementing and providing a foundation for them.

* Supported in part by NSF grant DCR 85-03275
The starting point for the development of operational semantics is a consideration of the important problem of what ideal machines are from a mathematical point of view. We do not find the existing solutions (VDL [8], LISP interpreters written in (a subset of) LISP [5], the SECD machine [4], and even Plotkin's transition systems [7]) satisfactory. The approach we shall describe, algebraic operational semantics, was originally proposed in [2]. To assess this new approach, we have worked out an algebraic operational semantics for the programming language Modula-2 (referred to subsequently as Modula) in its entirety. We have chosen Modula as our example because it is, in many ways, a model imperative programming language. It is largely free of extraneous constructs and integrates machine-dependent facilities in an elegant way. This paper gives a self-contained illustration of our approach using Modula as an example. A complete description of Modula is found in the Ph.D. dissertation [6]. Semantic accounts of Smalltalk and Occam using algebraic operational semantics are in preparation ([1] and [3], respectively).

So, what is an abstract machine for a programming language from a mathematical point of view? In algebraic operational semantics, it is an evolving (or dynamic) algebra (or structure) of a sort, tailored explicitly for the language at hand. What is a dynamic structure? Here we restrict ourselves to sequential evolving structures; in connection with distributive evolving algebras, see [3].

Each state of an evolving structure is what the logician would call a finite, many-sorted, first-order structure. It comprises a number of finite sets called universes and functions on Cartesian products of universes. (The presence of a Boolean universe allows one to treat relations as Boolean-valued functions; in that sense the static structures are algebras.) In the case of Modula, the signature (also called vocabulary or language) of the current state does not change during the structure's evolution, but some of the functions and universes may. Accounts of languages other than Modula may require a dynamic signature.

One distinguishing feature of algebraic operational semantics is that its universes are usually (finite and) bounded; in other words, its abstract machines usually have bounded resources. We do not view finite machines necessarily as approximations to infinite ones. For example, a machine equipped with genuine integers will loop forever executing

\[ n := 1; \text{WHILE true DO } n := 2 \times n \text{ END,} \]

but no machine with bounded resources will. The initial state of a dynamic structure should reflect all its resource bounds. Thus, given a particular programming language L, algebraic operational semantics defines a family of families of machines. The former are determined by programs written in L and, given a particular L-program Prog, the latter are determined by (the fragment of L used in Prog and) the resource bounds of the dynamic structures for Prog.

Transition rules guide the evolution of a dynamic structure from state to state. We shall give their syntax in a moment. A structure's transition rules should depend only on the language for which the structure provides semantics. Moreover, if the components of the structure have been chosen properly, the changes described by its transition rules should be slight.

For the purposes of this paper, we invoke the principle of separation of concerns and restrict our attention to the dynamic semantics of programs. Towards this end, we