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; WHILE true DO n := 2 * n 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

assume that a program is represented by its parse tree with respect to a given context-free grammar and that the initial state of an algebra reflects the static semantics of the given program.

1. The Syntax of Transition Rules

We begin with transition rules without parameters (free variables). The basic component of a transition rule is called an *update*. There are three sorts of update. Let S be a state of an evolving algebra M.

(i) Function updates: let f be a function symbol in the signature of M. Suppose that the type of f is $U_1 \times \ldots \times U_k \rightarrow U_0$ where each U_i is a universe name. Let $e_0, \ldots, e_k$ be closed (i.e. without free variables) expressions (terms) of types $U_1, \ldots, U_k$. Then $f(e_1, \ldots, e_k) := e_0$ is a function update. Its meaning is: first compute $e_0, \ldots, e_k$ in S and let $a_0, \ldots, a_k$ be the results, respectively; then assign a_0 to $f(a_1, \ldots, a_k)$. Read and write operations are treated as special forms of function update. A read operation is of the form $f(e_1, \ldots, e_k) :=$ *Input(channel)*, where f and the e_i are as before and *channel* is a path over which information passes. The meaning of a read operation is: when a value v is obtained from outside M over *channel* (in a given state S), evaluate $e_1, \ldots, e_k$ in S and let $a_1, \ldots, a_k$ be the results, then assign v to $f(a_1, \ldots, a_k)$. A write operation is of the form *Output(channel)* $:= f(e_1, \ldots, e_k)$ where f and the e_i are again as above. The meaning of a write operation is: compute $e_1, \ldots, e_k$ in the given state and let $a_1, \ldots, a_k$ be the results, then transmit $f(a_1, \ldots, a_k)$ outside M over *channel*.

(ii) Contractions of universes: let e be a closed expression, then *Dispose(e)* is a universe contraction. Its meaning is: compute e in 5 and let *a* be the result; then delete a from the universe to which it belongs. The deletion of a may make some functions undefined on some elements of their domains. The usual trick of using dummy elements "undefined", "uninitialized", etc. allows one to deal with total functions only.

(iii) Extensions of universes: let U be a universe name and F be a list of function updates some of which mention a variable *temp*, then

let *temp = New(U)* **in** F **endlet**

is a universe extension. Its meaning is: first add a new element to U and let *temp* name this element temporarily; then perform the function updates in the list F. The scope of *temp* is delimited by the brackets **let** and **endlet.**

Basically, a transition rule is of the form

if e **then** F **endif,**

where e is a closed Boolean expression and F is a list of updates. The meaning of such a transition depends on the value of e in the given state. If e is false, the rule

does not alter the state of the algebra; if e is true, the state of the algebra is altered according to the updates in F. The whole language is described by a finite set of transition rules which are executed simultaneously. A priori, different transition rules or even different updates of the same transition rule can contradict each other; we restrict our attention to (deterministic) consistent evolving algebras.

For brevity and convenience, we allow a slightly more complicated syntax. Let $r_1, \ldots, r_k$ be updates or transition rules and e be a closed Boolean expression, then

if e **then**
$r_1, \ldots, r_k$
endif

is a transition rule. Its meaning is: perform $r_1, \ldots, r_k$, if e is true in S and do nothing, otherwise.

There are several, tightly circumscribed, situations in a semantic account of Modula where parameterized transition rules are natural. Two such situations occur at block entry and block exit, when a relatively large number of locations must be allocated or deallocated. Modula specifies no ordering of the allocations or deallocations, hence, the natural thing to do is perform them "simultaneously". We express this simultaneity, or better, absence of ordering, by a parameterized transition rule. The meaning of such a rule is: perform the rule for all possible values of its parameters.

2. A State of a Modula Evolving Algebra

Since an evolving algebra for Modula reflects a given program, we give a sample program Prog which will allow us to supply concrete examples of the universes and functions comprising an algebra. The sample program appears in Figure 1.

We now describe the universes, functions and relations that comprise an evolving algebra *M*(Prog) for Prog. We also mention in passing those components which might be present in an algebra for a Modula program different from Prog but which are unnecessary to an account of Prog's ideal machine. We use *evolving algebra* and *dynamic structure* interchangeably in our account. All universes of a dynamic structure have the equality relation defined on them.

2.1. Integers

Prog declares a record type, Vertex, that includes a field of type integer. Therefore, a dynamic structure *M*(Prog) will include a universe *int* consisting of all the integers in

```
MODULE Prog;
FROM      Storage    IMPORT ALLOCATE;
FROM      InOut    IMPORT ReadInt, Done, WriteInt, WriteLn;
TYPE      Link = POINTER TO Vertex;
          Vertex = RECORD
                      datum: INTEGER;
                      left, right: Link
                   END;
VAR           r, tree: Link;

PROCEDURE Insert(item: Link; VAR subtree: Link);
BEGIN
 IF subtree = NIL THEN
   subtree := item;
   subtree ↑.left := NIL;
   subtree ↑.right := NIL
 ELSEIF item ↑.datum < subtree ↑.datum THEN
  Insert(item, subtree ↑.left)
 ELSE
  Insert(item, subtree ↑.right)
 ENDIF
END Insert;

PROCEDURE Print(subtree: Link);
BEGIN
 IF subtree ↑.left ≠ NIL  THEN Print (subtree ↑.left);
 WriteInt(subtree ↑.datum,6);Writeln;
 IF subtree ↑.right ≠ NIL THEN Print(subtree ↑.right)
END Print;

BEGIN
 tree := NIL;
 NEW(r); ReadInt(r ↑.datum);
 WHILE Done DO
  Insert(r, tree);
  NEW(r); ReadInt(r ↑.datum)
 END;
 IF tree ≠ NIL THEN Print(tree)
END Prog.
```

Figure 1. An Example Program

the interval [*MinInt*, *MaxInt*]. *MinInt* and *MaxInt* are distinguished elements of *int*. The universe *int* comes equipped with the usual ordering and the partial arithmetic operations +, −, ×, quotient, and remainder.

2.2. Boolean

A structure *M*(Prog) will include a universe *bool* = *{true, false}* equipped with the usual Boolean operations and ordered such that *false* < *true*. The binary Boolean operations are used *only* in transition rules. In Modula, the binary Boolean operations appear in a "conditional" form in which both an operation's arguments are not always evaluated. Consequently, their semantics are given by the transition rules that govern sequencing through the parse trees of Boolean expressions.

2.3. Other Basic Types

The other universes a Modula structure may comprise are a finite ordered set of characters *char*, an initial segment of the natural numbers *card*, and a finite set of real numbers *real*. In general, *char* and its ordering relation differ among Modula structures, but *char* must contain the upper case letters of the Roman alphabet, the digits 0, . . . , 9, and certain punctuation marks (cf. The Modula Report [9]). The universe *char* is equipped with operations, *Ord* and *Char*, which give the position in the ordering of one of its elements and the element corresponding to a position in the ordering, respectively. The universe *card* includes those natural numbers less than or equal to *MaxCard*, where *MaxCard* is a constant that differs among Modula structures. The usual ordering and (partial) arithmetic operations are defined on *card*. We omit a description of *real*.

2.4. The Parse Tree

The parse tree of Prog is represented by a universe called *parsetree* with a relatively rich structure plus some additional functions from and to *parsetree*. The elements of *parsetree* are the nodes of the parse tree. The partial functions *Child*1, *Child*2, . . . map an element *of parsetree* to its first, second, etc. child, if it has one. The function *Children* indicates how many children a node possesses. The function *Parent* maps an element of *parsetree* to its parent node, if it has one.

An auxiliary universe *grammarsymbol* provides labels for parse tree nodes which indicate the grammatical category to which the subtree under a node belongs. A function *Label : parsetree → grammarsymbol* represents this correspondence.

There is also a function *Sp* (for specification) that is used to simplify the representation in the parse tree of identifiers and constants. The grammar for Modula includes productions which describe the syntax of identifiers and constants. However, it is more convenient to deal with the identifiers themselves or the values denoted by the constants than with parse sub-trees representing their syntactic analysis. So we allow leaf nodes in our parse trees to be labeled by the non-terminal grammatical symbols 'id' and 'const' and have *Sp* map such a node to the appropriate identifier or value. Nodes labeled 'id' are mapped to a finite universe *ident* of identifiers; *ident* is equipped with the equality relation only. The values to which *Sp* maps constants are taken from universes such as *int*.

For convenience, a Modula evolving algebra comes equipped with a binary relation, *SubTree*, which, for a given parse tree node *n*, indicates which nodes are in the subtree of which *n* is the root. This relation is useful in certain transition rules, as we shall see in section 3.

The are two dynamic distinguished elements of *parsetree: AN* and *XN*. *AN*, for "active node", indicates at which node control currently resides. *XN* is an "auxiliary" active node which is used in the transition rules for declarations and procedure calls when there needs to be, in effect, two active nodes because two sub-trees must be traversed in synchronized fashion.

2.5. Raw Variables

Modula evolving algebras include a universe *rawvar* whose elements serve as the denotations of variable identifiers and, hence, play a central role in our account of program variables. However, we must first consider the semantic complication arising from the fact that Modula allows identifiers to be re-used. In our example program, 'subtree' appears in both the procedures Insert and Print. In order to give unique names to variables, procedures, etc., we adopt the convention of prefixing identifiers with the identifiers of the procedures and modules in which their declarations are nested in order from largest to smallest enclosing block. For example, the variable identifiers of Prog become InOut.Done, Prog.r, Prog.tree, Prog.Insert.item, Prog.Insert.subtree, and Prog.Print.subtree. The denotation of each of these extended identifiers is a unique element of *rawvar*, i.e. a raw variable. The denotations of extended identifiers are called raw variables because, in general, a block that is the body of a procedure may be activated recursively creating multiple incarnations of its variables each of which is a variable in its own right. More about this in a moment.

2.6. The Universe of Type Representatives

A Modula structure will include a universe *types* whose elements are tokens which represent the structure's data types. In particular, elements of *types* serve as the denotations of type identifiers. Our example program Prog uses the following types; for each of them, *types* contains a distinct element. INTEGER appears as the type of one of the fields of the record type Vertex; BOOLEAN appears implicitly as the type of the imported variable Done and certain expressions; Vertex is a declared record type; Link is declared of type 'POINTER TO Vertex'. Each of the procedures InOut.ReadInt, InOut.WriteInt, InOut.WriteLn, Prog.Insert, and Prog.Print is of a different procedure type (determined by the types of their parameters and whether they're value or variable) and corresponds to an element of *types*.

2.7. Locations

Modula structures include a universe *loc* whose elements play a dual role. First, they represent the incarnations of raw variables produced by activating the blocks in which the raw variables are introduced. Secondly, they represent the elements of dynamic data structures created by the procedure NEW. The latter role requires that *loc* be a dynamic universe, since the number of calls on NEW to be expected during the execution of a program cannot, in general, be predicted. Since calls on NEW produce locations, it should be apparent that locations are the values which incarnations of pointer variables assume. For example, Prog introduces a number of raw variables of type 'POINTER TO Vertex'; each of these will be mapped, by functions to be described below, to locations which will, in turn, be mapped to values of a universe corresponding to Vertex. Note that our locations are more abstract than those sometimes appearing in theories of programming language semantics. They are not intended to model an ideal computer's "memory": they are not ordered and there are no operations other than the equality relation defined on them. There is no notion of "re-using" locations in our semantics; new locations are created and old locations are dropped, but the story ends there—the new locations created bear no relation to the old ones dropped. However, our structures are *finite* and, moreover, bounded, therefore every structure places a limit on the size of *loc*.

2.8. Structured Data Types

Our example program Prog introduces a record type Vertex. For every record type, there exists a dynamic universe which contains an element for every instance of the

record type created but not yet deleted at that point. In the case of Prog, let *vertex* refer to the universe of *M*(Prog) corresponding to the record type of the same name. The denotations of the field names of the record type Vertex will be dynamic functions from *vertex* → *loc:* the function corresponding to 'datum' will map elements of *vertex* to locations that assume integer values and the functions corresponding to 'left' and 'right' will map elements of *vertex* to locations that assume values that are themselves locations. These functions are dynamic because transition rules expanding their domains must be applied to them as elements are added to *vertex*. Note that, for each element added to *vertex*, three elements are added to *loc*.

We next consider arrays even though our sample program includes no array types. To an array type corresponds a universe, initially empty, whose elements represent instances of the array type, and an "access" function, which maps pairs consisting of an instance of the array type and an index value to a location. To create a new instance of an array type, one adds a new element to the appropriate universe of array values, one adds new locations to contain the array's components, and one applies a transition rule to the array type's access function to cause it to map pairs consisting of the new array value and an index value to the corresponding new locations.

The components of arrays and records are represented as locations because they may be passed as actual parameters to procedures with variable formal parameters. This means that an array or record component may become an "incarnation" of the raw variable that is the denotation of a variable procedure parameter and the incarnation of a raw variable is a location.

2.9. Command Results and Space

Modula dynamic structures include a universe *result* comprised of three elements: *ok*, *error*, and *uneval* (for unevaluated). The elements of *result* are used to signal the outcome of sub-computations, such as those specified by Modula commands, that don't otherwise produce a result or to indicate that control has yet to visit a sub-parse-tree.

Each dynamic structure will include a universe *space* whose elements are called *units*. What a unit corresponds to varies among structures. A unit may correspond to a bit, a byte of 8 bits, or a word of some number of bits. For each data type a program introduces, a function *Size* tells us how many units of *space* correspond to that type. A Modula dynamic structure will also include a dynamic function *Avail* that indicates, at any given moment, how much space is available.

2.10. The Static Functions *Intro, Sig,* and *Type*

Modula allows the reuse of identifiers. However, the declaration or procedure

parameter specification introducing the identifier that is in force at any point in the program may be determined by examination of the program's text. Therefore, every identifier occurring in a Modula program can be uniquely associated with an introduction of the identifier. Moreover, this association may be established without executing the program. Consequently, we assume that Modula structures come equipped with a static function, *Intro,* which maps an identifier node in the parse tree to the node representing the same identifier in the appropriate declaration or parameter specification subtree introducing the identifier. For example, in the case of Prog, *Intro* maps occurrences of 'r' in the main body to the occurrence of 'r' in Prog's variable declaration list. It maps occurrences of 'subtree' in the body of Insert to the occurrences of 'subtree' in Insert's formal parameter list and occurrences of 'subtree' in the body of Print to the occurrence of 'subtree' in Print's formal parameter list. *Intro* also maps procedure identifier nodes to the root of the procedure's declaration subtree.

Modula structures also include a static function *Sig* which maps nodes representing defining occurrences of identifiers to their significations. In particular, *Sig* maps identifier nodes in variable declarations or formal parameter specifications to the raw variable that is the denotation of the variable or parameter. Hence, given a variable identifier node, one obtains that variable's denotation by applying the composition of *Intro* and *Sig* to the identifier node. For example, if *n* is a node labeled 'id' in the subtree for Insert whose specification is the identifier 'subtree', then *Sig*(*Intro*(*n*)) is the raw variable Prog.Insert.subtree. In principle, the function *Sig* is unnecessary − raw variables can be identified with the corresponding nodes of the parse tree. However, since raw variables play such an important role, we find it convenient to distinguish them from the corresponding nodes of the parse tree.

Modula structures include a static function *Type* which maps a parse tree node in the range of *Intro* to the element of *types* representing the type of the object introduced at the node.

2.11. The Dynamic Function *Top* and The Predecessor Relation on *loc*

We have seen how the static functions *Intro* and *Sig* take us from a program variable node to the raw variable that is the program variable's denotation. The possibility of recursively activating the block in which a program variable is introduced means that multiple incarnations of the raw variable may exist in some state of a dynamic structure. The problem is to keep track of these incarnations in such a manner that the value of the most recently created one is fetched when needed and that the previous incarnation is restored when control leaves the block in which the program variable was introduced. Consider also the following sort of "variables" which are implicitly part of a Modula structure. In our semantics, a subcomputation ideally corresponds to a traversal of a parse tree. During such a traversal partial results are produced. These partial results are made available by "attaching" them to appropriate parse tree nodes. In this scheme, a parse tree node may be thought of as

corresponding to an implicit variable whose value is the result of performing the sub computation represented by the subtree under the node. The possibility of recursion means that these implicit variables may have multiple incarnations. Therefore, the problems of coping with incarnations mentioned above obtain with them as well. Modula structures include two dynamic functions which solve these problems. The first is *Top,* which maps raw variables and parse tree nodes to locations. Specifically, *Top* maps a raw variable to the location that represents its most recent incarnation and it maps a parse tree node to the most recent incarnation of the implicit variable corresponding to the node. Given a parse tree node *n* representing a variable, we obtain the most recent incarnation of that variable by applying the composition of *Intro, Sig,* and *Top* to *n*: *Top*(*Sig*(*Intro*(*ri*))). To obtain the most recent incarnation of the implicit variable corresponding to a node, one applies *Top* to the node directly.

When control leaves the block in which a variable is introduced, a transition rule must be applied to *Top* to restore its previous value at the (raw or implicit) variable, if it had one. To remember previous incarnations of variables, Modula structures include a partial dynamic function *Pred.* When applied to a location and raw variable or parse tree node, *Pred* yields the location representing the raw or implicit variable's previous incarnation, if it has one, and is undefined otherwise. *Pred* takes a raw variable as well as a location as argument because it is possible, via aliasing, for a single location to represent an incarnation of more than one raw variable and, hence, to have different predecessors depending on which raw variable one considers. This situation occurs when a program variable is passed as actual parameter in a procedure call for a formal variable parameter.

2.12 The Dynamic Function *Val*

Modula structures come equipped with a dynamic function *Val* which assigns values to locations. The range of *Val* includes those data types used in a program. In the case of Prog, we have:

$$Val : loc \rightarrow int \cup bool \cup loc \cup vertex.$$

For convenience, in the transition rules, we let *Nval*(*n)* abbreviate *Val*(*Top*(*n*)), where *n* is an implicit variable. *Nval*(*n*) always gives the value of the most recent incarnation of the implicit variable.

2.13 The Procedure Stack

The procedure stack consists of a dynamic universe *pstack* with dynamic distinguished element *PSTop,* a function *PStack : pstack* → *parsetree,* and a relation on

pstack, *PSPred* (for predecessor). *PSTop* is the "top" element of the stack, *PStack* maps each element of the stack to the root of a procedure call subtree, and *PSPred* records the history of yet-to-be-completed procedure calls.

3. Some Representative Transition Rules

First, we describe the transition rule for assignment statements. The grammar production describing assignment statement subtrees is

assignment → desig := exp.

The semantics of assignments are familiar. Evaluate the designator on the left to obtain a location *l*; evaluate the expression on the right to obtain a value *v*; make *v* the new value of the dynamic function *Val* at *l*. However, the transition rule for assignments is somewhat complicated by the requirement that one transition rule suffice for all instances, in a parse tree, of a particular grammatical category. Specifically, expressions may appear in a number of contexts; among them are assignment statements and actual parameter lists of procedure calls. There exists a potential conflict between the kind of value required in these two contexts. An expression appearing in an assignment statement should always evaluate to an expression value (sometimes called an *r-value*). An expression appearing in place of a variable formal procedure parameter should always evaluate to a location (sometimes called an *l-value*). Most expressions pose no problem: if the expression contains operators (other than array indexing, record field selection, and pointer dereferencing) it will always evaluate to an r-value. If the expression consists only of a variable, the transition rules must cause it to evalute to a location. Then, if the expression's context requires an l-value, a location is available; if the expression's context requires an r-value, the location can be coerced to an r-value by an application of *Val*. An auxiliary function *Value* performs the coercion:

$$Value(x) = \begin{cases} Val(x), & \text{if } x \in loc; \\ x, & \text{otherwise.} \end{cases}$$

The transition rule for assignment statements appears in figure 2.

Next, we give the transition rule for procedure call subtrees. The relevant grammar production is:

procall → id (explist).

When *AN* is labeled 'procall', its first child is the procedure's identifier and its third child is the subtree representing the formal parameter list. The transition rule for 'procall' subtrees consists of three inner transition rules. The first transfers control to the formal parameter list subtree if it has not been evaluated. The second invokes three actions: it updates the procedure stack, it allocates a new location for the implicit variables corresponding to the nodes of the subtree for the procedure being called, and it transfers control to the procedure. The transition rule that creates new incarnations of

```
if Label(AN) = assignment then
 if Nval(Child1(AN)) = uneval then
  AN := Child1 (AN)
 endif,
 if Nval(Child1 (AN)) ≠ uneval then
  if Nval(Child3(AN)) = uneval then
   AN := Child3 (AN)
  endif,
  if Nval(Child3 (AN)) ≠ uneval then
   Val (Nval(Child1(AN))) := Value (Nval(Child3 (AN))),
   AN := Parent (AN),
   Nval(AN) := ok,
   Nval(Child1(AN)) := uneval,
    Nval(Child3(AN)) := uneval
  endif
 endif
endif.
```

Figure 2. The Transition Rule for Assignment Statements

the procedure's implicit variables is an example of a parameterized transition rule. Its parameter *n* ranges over the nodes in the 'block' subtree that constitutes the body of the procedure being called. When the active node is the root of the procedure call, the root of the subtree for the body of the procedure is given by '*Child3*(*Intro*(*Child1*(*AN*))'. The transition rule that updates the procedure stack extends the universe *PStack*. The element added is subsequently removed, by the update *Dispose* (*PSTop*), when the called procedure is exited. The third inner rule transfers control to the procedure call's parent after the call has completed. Note here that completion of the call is indicated by '*Nval*(*AN*) ≠ *uneval*'. The value of this implicit variable is changed just prior to exiting the body of the procedure. The transition rule for 'procall' nodes is given in Figure 3.

4 An Application

In this section we show how our methods of semantic definition may be extended to certain "low-level" facilities of Modula and then indicate how the correctness of a simple keyboard interrupt handling routine can be proven. In doing so, we shall have to incorporate the interrupt mechanism and input/output channels of a hypothetical computer into our model. We hope to accomplish two purposes: first, to illustrate how connections between semantic models of "official" Modula and its implementations can be made and, second, to show how our semantic models may be used to prove properties of Modula programs. The keyboard interrupt handler we shall use as an example is taken from [9]. It is presented in Figure 4. This program fragment is based on a PDP-11 implementation of Modula, although we do not claim to have formalized the PDP-11 here. Rather, we formalize those properties of an underlying machine required to reason

```
if Label(AN) = procall then
 if Nval(AN) = uneval and Nval(Child3(AN)) = uneval then
   AN := Child3(AN)
 endif,
 if Nval(AN) = uneval and Nval(Child3(AN)) ≠ uneval then
  if Subtree(n, Child3(Intro (Child1(AN)))) then
    let temp = New (loc) in
      Val (temp) := uneval
      Top(n) := temp
      Pred(temp, n) := Top(n)
    endlet
  endif,
  let temp = New (pstack) in
    PStack(temp) := AN,
    PSPred(temp) := PSTop,
    PSTop := temp
   endlet,
   AN := Intro(Child1(AN))
 endif,
 if Nval(AN) ≠ uneval then
   AN := Parent(AN)
 endif
endif.
```

Figure 3. The Transition Rule for Procedure Calls

about the program Wirth presents in [9]. We have chosen an interrupt handling routine as our example program because we believe it demonstrates the utility of our approach to semantics most effectively. The very notion of interrupt involves the concept of time: an interrupt is an event which occurs at an arbitrary *moment* in the evolution of a computation. And the idea that a computation is a sequence of states unfolding in time is the basis of operational semantics. Moreover, the state changes which an interrupt engenders are not directly connected to any part of a program's text. Therefore, a semantic theory which ascribes, for example, mathematical functions to components of program text will not deal readily with interrupts. Yet, interrupts are fundamental, at the right level of detail, to the function of all modern computing machinery.

We shall now describe those aspects of our example program which are not part of "official" Modula, i.e. the low-level facilities of which it makes use. The first is the notion of process in general and coroutine in particular. Implementations of Modula are free to adopt a concept of process of the designer's choice. Obviously, this choice will be largely, but not entirely, determined by the hardware on which the implementation is to run. On single processor machines, the coroutine concept is attractive. We shall restrict our attention to coroutines. The basic idea of a coroutine is the quasi-concurrent execution of a number of sequential processes, i.e. at any given moment only one of two or more sequential processes is executing. An executing process may suspend itself and transfer control to another, which then resumes executing where it last left off. Each process has its own local state information as well as (possibly) access to data structures shared

```
MODULE keyboard[4];
  EXPORT fetch, n;
  IMPORT ADR, SIZE, WORD, PROCESS,
         NEWPROCESS, TRANSFER, IOTRANSFER;
  CONST  N = 32;
  VAR    x[777562B] : CHAR; (* keyboard data *)
         s[777560B] : BITSET; (* keyboard status *)
  VAR    n, in, out : CARDINAL;
         buf : ARRAY[0..N - 1] OF CHAR;
         PRO, CON : PROCESS;
         wsp : ARRAY[0..177B] OF WORD;

  PROCEDURE  fetch(VAR ch : CHAR);
    BEGIN (* to be called only if n > 0 *)
      IF n > 0 THEN
        ch := buf[out]; out := (out + 1) MOD N;
        n := n - 1
      ELSE ch := 0C
      END
    END fetch;

  PROCEDURE producer; (* acts as coroutine *)
    BEGIN
      LOOP
        IOTRANSFER(PRO, CON, 60B);
          IF n < N THEN
            buf [in] := x; in := (in + 1) MOD N;
            n := n+ 1
          END
        END
      END producer;

  BEGIN
    n := 0; in := 0; out := 0;
    NEWPROCESS (producer, ADR(wsp), SIZE(wsp), PRO);
    EXCL(s, 6); TRANSFER(CON, PRO)
  END keyboard.
```

Figure 4. A Keyboard Handler Module (from [9])

with its coroutines. Since coroutines are considered low-level facilities, their associated data type and its operations are imported from the module SYSTEM; the import list in our example program reflects this fact. A process is determined by a parameterless procedure which must be declared in the outermost block of a module. A process may be thought of as an instantiation, created by the pre-defined procedure NEWPROCESS, of the procedure which determines it. This means, among other things, that the process will have its own copies of all the procedure's local data structures. NEWPROCESS is exported by the SYSTEM module. Control is explicitly transferred to and from a process by means of predefined procedures TRANSFER and IOTRANSFER which are also exported from the SYSTEM module. The process in which a call on NEWPROCESS is executed becomes the parent of the created process.

Our example program imports the data type PROCESS from the SYSTEM module. As its name suggests, elements of this data type represent processes. To provide a denotation for this data type, we augment our dynamic structure with a dynamic universe *processes*. The universe of processes has a dynamic distinguished element *AP* (for active process) which indicates which process is currently executing. In the initial state of a dynamic structure *processes* will contain a single element. Elements are added to and deleted from *processes* as processes are created and deleted. As mentioned above, processes are created by calls on the procedure NEWPROCESS. A process is deleted when control reaches the end of the procedure which determines it or when control reaches the end of the procedure which determines one of its parent processes. Since each process must have its own local state space, we must alter the dynamic functions *PSTop*, *Top*, and *Avail* so that they take as additional argument an element of *processes*. Similarly, the dynamic distinguished elements of *parsetree* − *AN and XN* − must now become dynamic functions from *processes* into *parsetree*.

The declarations of the variables 'x' and 's' in our example both include an octal constant (777562B in the case of 'x' and 777560B in the case of 's' − the 'B' indicates that the preceding digits are to be interpreted as octal digits) which is meant to be interpreted as a memory address. This is so because the input/output registers of our hypothetical computer are "memory-mapped", i.e. one refers to them as if they were memory cells, and, in our example, we wish to use the names 'x' and 's' to refer to the keyboard data and status registers, respectively. Thus, the declarations of 'x' and 's' must indicate the address of the registers with which they are to be associated. In our model 'x' and 's' will be bound to input channels. This is accomplished by adding a new symbol 'ioregister' to *grammarsymbol* to represent such variables and adding new grammatical productions 'factor → ioregister' and 'ioregister → id'. The range of *Sig* must be expanded to include input/output channels. The value of *Sig* at the 'id' node in the declaration for a variable bound to a channel will be initialized to the name of the channel to which the variable is bound. In the case of 'x', for example, this name is the octal constant '777562B'. We give the transition rule for the production 'ioregister → id' when the 'id' node refers to the keyboard data register

of our hypothetical computer:

if *Label*(*AN*(*AP*)) = ioregister and *Sig*(*Intro*(*Child*1(*AN* (*AP*)))) = 777562B
then
Nval(*AN* (*AP*)) := *Input* (*Sig*(*Intro* (*Child*1 (*AN* (*AP*))))),
Output(777560B) := *Input*(777560B) - {6},
AN (*AP*) := *Parent*(*AN* (*AP*))
endif.

This function update '*Output* (777560B) := *Input* (777560B) - {6}' reflects the fact that bit 6 of the keyboard status register of our hypothetical computer is reset when its keyboard data register is read. (The keyboard status register is declared to be of type BITSET and the operator '-' denotes set difference here.)

The pre-defined procedure NEWPROCESS has as its heading:

```
PROCEDURE NEWPROCESS (P : PROC, A : ADDRESS,
                      n : CARDINAL, VAR new : PROCESS).
```

In a call on NEWPROCESS, the actual parameter corresponding to 'P' denotes the procedure that determines the process to be created, the actual parameter corresponding to 'A' gives the base address of the workspace in which the processes' local variables are to allocated, the actual parameter corresponding to 'n' gives the size of this workspace, and the actual parameter corresponding to 'new' is a variable of type PROCESS in which a reference to the created process is stored. PROC is a pre-defined data type "parameter-less procedure". At the level of abstraction at which we are formalizing our example program we will not need the base address of the new process' workspace. Therefore, we may omit a discussion of how our hypothetical computer's memory is modeled. Calls on NEWPROCESS are characterized by the CFG production

predefcall → id(explist),

where the specification of the node labeled 'id' is 'NEWPROCESS'. The corresponding transition rule first prescribes evaluation of the actual parameters of the call. In what follows let 'P', 'new' and 'n' denote the roots of the subtrees in the actual parameter list of a call on NEWPROCESS corresponding to the formal parameters of the same names. A suite of function updates is applied to *Val*, *Avail*, and *AN*. A function update '*Val* (*Nval*(*new*)) := *temp*' sets the value of the most recent incarnation of the actual parameter corresponding to 'new' to the newly created element of *processes*; this makes this parameter into a reference to the new process, as desired. A function update '*Avail*(*temp*) := *Value*(*Nval*(*P*))' sets the amount of available storage for the new process. A function update '*AN*(*temp*) := *Value*(*Nval*(*P*))' sets the active node for the new process. Note that this update does not activate the new process, since the current active process (indicated by *AP)* is the process executing the call on NEWPROCESS; control remains in this latter process and proceeds to the parent node of the call on NEWPROCESS. The root of the procedure call subtree is marked *ok* to signal successful creation of the new process.

The predefined procedure TRANSFER has the following heading:

```
PROCEDURE TRANSFER (VAR source, destination : PROCESS);
```

In a call on TRANSFER, the actual parameter corresponding to 'source' will be a variable of type PROCESS in which a reference to the process executing the call, i.e. an element of *processes*, will be stored; the actual parameter corresponding to 'destination' will be a variable of type PROCESS in which a reference to the process to be activated is stored. Calls on TRANSFER are described by the same CFG production as calls on NEWPROCESS. In the following let 'source' and 'destination' denote the roots of the subtrees in the actual parameter list of a call on TRANSFER corresponding to the formal parameters of the same names. The function update *'AP* := *Val*(*Nval*(destination))' activates the process represented by the value stored in the variable supplied as second actual parameter of the call on TRANSFER. This actual parameter must evaluate to a location (since it corresponds to a VAR formal parameter), hence the appearance of *Val* in the function update. The function update '*Val*(*Nval*(source)) := *AP'* stores a reference to the current process in the location to which the first actual parameter of the call on TRANSFER evaluated. The two remaining function updates advance control in the about-to-be-deactivated current process to the parent of the call on TRANSFER and mark the root of procedure call subtree *ok*. When this process is later reactivated, execution will resume at the parent of the call subtree.

Before we describe the semantics of calls on the predefined procedure IOTRANSFER, we must describe our model of interrupts. An interrupt is essentially the communication, from outside a dynamic structure, of a value of type CARDINAL. This communication takes place over a channel *interrupt*. The specific CARDINAL value communicated indicates which agent initiated the interrupt; the assignment of values to agents depends on the configuration of the computing system being modeled. Each agent has a priority. Priorities will also be expressed by CARDINAL values. For this purpose we add a static function *IntPriority* to our dynamic structure. *IntPriority* maps a CARDINAL value representing an interrupting agent to the CARDINAL value that expresses its priority: the priority of agent *I* is greater than that of agent *j*, if *IntPriority* (*i*) > *IntPriority* (*j*). Each interrupting agent will have associated with it a process called its *handler*, which is activated, in a manner to be described shortly, whenever the agent interrupts. The element of *processes* that represents a handler will be stored in a location which is obtained by applying a function *IntHandler* to the CARDINAL value representing the agent. For each interrupting agent we must also reserve a location to hold the representative of the process that was executing when the agent interrupted. This is so that control may be returned to this process when the agent's handler has completed its job. We obtain this location by applying a function *IntRetDes t* to the CARDINAL value representing the agent. For a particular interrupt *i*, the value of *IntPriority* (*i*), *IntHandler*(*i*), and *IntRetDest*(*i*) collectively constitute the *interrupt vector* for *i*.

We must extend the notion of priority to the statements of a Modula program. We shall do so by adding to our dynamic structure a dynamic function *CurPriority* which maps elements of *processes* to CARDINAL values: for each process, *CurPriority* gives the current priority level of the statement executing in that process. Now, how is the current priority established? Note that heading of the declaration of the module 'keyboard' in our example program includes the symbol '[4]'. This assigns a priority of 4 to all the executable statements of the module and its procedures. Otherwise, the statements of a procedure inherit the priority of the program that called the procedure. We represent this situation as follows. To deal with modules with an explicitly declared priority (like 'keyboard' in our example),

we augment our dynamic structure with a static, partial function on *parsetree*, *ModulePriority*, which maps all the nodes of such a module to the CARDINAL value that expresses the module's priority. Whenever one of the module's procedures is called, *CurPriority* is set to the value obtained by applying *ModulePriority* to the root of the procedure's declaration subtree. In order to restore the priority of the calling program, we stack the value of *CurPriority* which prevailed during its execution. To accomplish this we need a dynamic function *PriorityStack* which maps elements of *pstack* to CARDINAL values. Moreover, for those procedures that are not part of modules with an explicit priority, *CurPriority* may be set to *PriorityStack* (*PSTop*), i.e. such procedures inherit the calling procedure's priority. The transition rules for 'procall' and 'procdecl' nodes must be modified. The interested reader may consult [6] for details.

The predefined procedure IOTRANSFER has the following heading

PROCEDURE IOTRANSFER(VAR source, dest : PROCESS; va : CARDINAL);

IOTRANSFER is like TRANSFER in that it activates the process whose representative is stored in the actual parameter corresponding to 'dest' and stores the current process's representative in the actual parameter corresponding to 'source'. In addition, it sets the interrupt priority, interrupt handler, and interrupt return destination attributes of the interrupt designated by the value of the actual parameter corresponding to 'va'. In the following let 'source', 'dest' and 'va' denote the nodes of the call subtree corresponding to the formal parameters of the same name. The semantics of IOTRANSFER are expressed by a number of function updates. In the following discussion, we mean by "the interrupt" the interrupt denoted by the value of the actual parameter corresponding to 'va'. The function update

$$IntPriority(Value\ (Nval(va))) := ModulePriority\ (AN\ (AP))$$

sets the priority of the interrupt to the priority of the call-statement. The function update

$$IntHandler(Value\ (Nval(va))) := Nval(\text{source})$$

sets the interrupt's handler to process represented by the value in the location to which 'source' evaluates. The function update

$$IntRetDest(Value\ (Nval(va))) := Nval(dest)$$

establishes the contents of the location to which 'dest' evaluates as the process to which control returns when the interrupt's handler has finished executing. We shall see next how an element of *processes* is stored in this location.

We shall now give the transition rule that describes how our dynamic structure changes when an external agent interrupts. When an external agent interrupts, an appropriate CARDINAL value is communicated to our dynamic structure over the channel *interrupt*. The interrogation of this value is indicated in a transition rule by writing '*Input* (*interrupt*)'; this term is undefined in a particular state of a dynamic structure, if no value is present on the channel. Therefore, the occurrence of an interrupt causes the guard '*Input* (*interrupt*) $\neq \perp$' to evaluate to *true*.

We then have the following transition rule for interrupts:

if *Input* (*interrupt*) ≠ ⊥ **then**
  if *IntPriority*(*Input* (*interrupt*)) > *CurPriority*(*A P*) **then**
   AP := *Val*(*IntHandler*(*Input*(*interrupt*)))
   Val(*IntRetDest*(*Input*(*interrupt*))) := *AP*,
  endif
endif.

Note that an interrupt is ignored if its priority is lower than the priority of the currently executing process. When an interrupt of sufficiently high priority occurs, a TRANSFER operation is effectively performed. The difference is that, in the case of an interrupt, there is no associated program text and, hence, no need to alter the active node or *Nval*. In order to make our dynamic structure deterministic, we must embed the transition rules we gave in Chapter 4 in an outer transition rule whose guard is

$$Input\ (interrupt) = \perp.$$

That is, in the absence of interrupts processing precedes as we have described it in previous sections of this paper.

Let Prog be a program that includes and uses the module 'keyboard' given above. How can one prove that the module 'keyboard' works correctly, i.e. that the characters fetched by Prog are exactly the characters entered from the keyboard. Here is one way. Define in the natural way

(a) sequences of characters *deposit-sequence, fetch-sequence,* and *buffer-contents,* and

(b) terms *in, out, n,* and *N* with values of type CARDINAL and establish that in all appropriate states

(1) *deposit-sequence* is the concatenation of *fetch-sequence* with *buffer-contents,* and

(2) *in = out + n Mod N.*

The reader is referred to [6] for details.

References

1. Blakley, Robert and Gurevich, Yuri, "The Algebraic Operational Semantics of Smalltalk", in preparation.

2. Gurevich, Yuri, "Logic and the Challenge of Computer Science", in Current *Trends in Theoretical Computer Science* (ed. E. Börger), Computer Science Press, 1987, 1-57.

3. Gurevich, Yuri and Moss, Lawrence, "The Algebraic Operational Semantics of Occam", in preparation.

4. Landin, Peter J., "A A-calculus approach", in Advances in Programming and Non-numerical Computation, L. Fox(ed.), London, Pergamon Press, 1966.

5. McCarthy, John and others, Tie Lisp *1.5 Programmer's Manual*, Cambridge, Massachusetts, MIT Press, 1962.

6. Morris, James, *Algebraic Operational Semantics for Modula 2*, Ph.D. dissertation, The University of Michigan, 1988.

7. Plotkin, Gordon, *A Structural Approach to Operational Semantics*, DAIMI FN-19, Aarhus University.

8. Wegner, Peter, "The Vienna Definition Language", *ACM Computing Surveys*, 4(1), 5-63.

9. Wirth, Niklaus, *Programming in Modula-2*, Berlin, Springer-Verlag, 1982.