An Executable Course in the Algebraic Semantics of Imperative Programs

Grant Malcolm and Joseph A. Goguen
Programming Research Group, Oxford University

1 Introduction

This paper describes a novel approach to teaching formal methods in Computing Science through an *executable* course on the semantics and verification of imperative programs. This approach allows students to learn both formal semantics and techniques for formal reasoning about programs by experimenting at a computer. Our experience teaching a course titled Algebraic Semantics of Imperative Programs suggests that this can significantly enhance learning.

This course has developed over the last seven years, during which some form of it was taught once each year; this development has recently culminated in a textbook [9]. The course is taught to graduate students from a variety of academic backgrounds beginning their serious study of Computing Science through the "conversion" MSc in Computation programme at Oxford; it could equally well be taught to undergraduates as early as their second year. The philosophy of our course and of the programme of which it is part is captured in the following quote from Christopher Strachey, founder of the Programming Research Group at Oxford:

> It has long been my personal view that the separation of practical and theoretical work is artificial and injurious. Much of the practical work done in computing, both in software and in hardware design, is unsound and clumsy because the people who do it have not any clear understanding of the fundamental design principles of their work. Most of the abstract mathematical and theoretical work is sterile because it has no point of contact with real computing. One of the central aims of the Programming Research Group as a teaching and research group has been to set up an atmosphere in which this separation cannot happen.

Specific goals of our course include the following:

1. improve intuition and ability in imperative programming, through understanding the semantics of programs and seeing numerous examples;
2. teach how to prove properties of programs;
3. develop relevant mathematical background;
4. present the OBJ system and use it for all proofs; and
5. show that all this can be done in a way that is completely rigorous, yet not too difficult or too abstract, by using equational logic, which is simply the logic of substituting equals for equals.

Our semantic approach gives an equational specification for a class of abstract machines and specifies programming language features by their effect on these machines. The features treated in the course are assignment, sequential composition, conditional, while-loop, procedure definition, and procedure call; we also give semantics for the array data structure and for programs that manipulate arrays. The equational axioms describing these features are used in proving program correctness.

Thus, this course both presents a rigorous semantics for imperative programs, and uses that semantics to prove properties of programs. Most comparable courses concentrate on one or the other of these aims, either presenting mathematical theories of semantics of programming languages, or presenting a logical calculus (such as Hoare-triples or weakest preconditions) and using this to prove properties of programs. In both cases, the course work is done on paper, since the semantics of programs and proofs of their properties are presented on paper.

By contrast, our Algebraic Semantics of Imperative Programs course uses an *executable* presentation of program semantics in OBJ [11], an implemented specification language that can be used for theorem proving and rapid prototyping, and which itself has a formal semantics based on equational logic. OBJ is not just another functional programming language, although it does have an executable functional sublanguage. OBJ was designed for algebraic semantics; its declarations introduce symbols for sorts and functions, its statements are equations, and its computations are equational proofs. OBJ also has a sublanguage of "theories" for declaring properties. Thus, an OBJ "program" actually *is* an equational theory, and every OBJ computation actually *proves* some theorem about such a theory. This means that an OBJ program used for defining the semantics of a program already has a precise mathematical meaning. Moreover, standard techniques for mechanising equational reasoning can be used for verifying programs.

This novel approach has several advantages:

- because OBJ has a formal semantics, our presentation of the semantics of imperative programs is mathematically rigorous;
- because OBJ's semantics is based on equational logic, which is simply the logic of substituting equals for equals, our presentation of the semantics of imperative programs is simple and readily understandable;
- because OBJ supports the specification of abstract data types, we can present the semantics of imperative programs that manipulate a variety of data structures.

However, the chief advantages of this approach stem from the most innovative aspect of the course: that the semantics is executable. This means that students can:

- acquire a feeling for the semantics of imperative programs by experimentation – they can use OBJ to 'step through' a computation and examine its results;
- acquire a feeling for the semantics of correctness proofs by experimentation – they can use OBJ and its semantics to construct and execute such proofs; and
- experiment with extensions of the semantics to treat features of imperative programs not directly covered in the course.

The next-to-last point represents the bulk of the course work for the students; each week they are given practical exercises which require them to construct proofs in OBJ. As new features of imperative programs are introduced in the lectures, they are encouraged to experiment with the semantics to consolidate their grasp of the material. This seems to be especially useful in learning the semantics of programs that manipulate unfamiliar data structures such as arrays (in previous years, the course has covered stack-based programming); students can gain insight into a data structure by observing the effect of running simple programs. Some of the exercises also introduce new language features and/or data structures, and ask the students to use them in programs and proofs.

We have found that this novel approach appeals to students, and that executability can greatly facilitate teaching an introductory course on formal semantics and formal reasoning about programs.

The following section gives a more detailed account of the background and goals of the course, and Section 3 gives an example of the material taught in the course so that readers can evaluate the claims we have made. The example consists of an OBJ theory for a class of abstract machines, the syntax and semantics of a simple programming language with while loops, and a correctness proof for a program that computes the exponential of two numbers. In fact, this example, which fits into a handout of three pages, constitutes the bulk of the introductory lecture to the course! Students are generally impressed that so much can be done so rigorously in so little space, and we hope the reader will share something of the excitement we still feel in presenting this material.

2 Some Goals and Issues in Algebraic Semantics

All formal methods rely on some semantics of computation. In recent years, many promising formal methods have arisen from algebraic approaches to semantics. We believe that algebraic approaches tend to be simpler and more tractable, in the sense that they lend themselves readily to formal reasoning, for example in proving the correctness of refinement or implementation [8, 19], or, as in our course on algebraic semantics, in providing a simple yet rigorous foundation for proving correctness of imperative programs. In this section we sketch the main differences between our algebraic semantics, which might be called *algebraic denotational semantics* (*ADS*), and other well established approaches.

Roughly speaking, the most popular approaches to the semantics of imperative languages divide into three groups: operational, axiomatic and denotational. An *operational semantics* describes the meaning of a programming language by describing a way of executing its programs. Often this is done by giving an interpreter or compiler for the language. So-called *structured operational semantics* (also called *natural semantics*) [22] describes computations by giving formal rules of deduction for steps of computation; this may be considered an abstract interpreter.

In *axiomatic* approaches, programming language features are defined by writing axioms in some logical system. First order logic, or some variant of it, is the most popular, since this is the logical system most widely used in mathematics and its foundations. An axiomatic approach using assertions and invariants was pioneered by Alan Turing [26] and John von Neumann [12], and later made more formal by Robert Floyd [5] and Tony Hoare [15].

By contrast, *denotational* approaches build mathematical models of programming language features; these models are called *denotations*. For example, the denotation of a program might be a partial function from inputs to outputs. Usually set theory is used in constructing these denotations, perhaps with some technical constraints, such as continuity. In this approach, the denotation of a program is constructed by composing the denotations of its parts. Denotational semantics was pioneered by John McCarthy [20], and greatly extended by Christopher Strachey and Dana Scott [24].

In classical denotational semantics (often called *Scott-Strachey semantics*, see [25, 13]), the model of storage can be criticised for being too concrete. However, ADS *axiomatises* the notion of storage, so that any model satisfying the axioms can be used, thus avoiding the need to select one particular model; that is, ADS uses so-called "loose" algebraic semantics to specify a *class* (i.e., "variety") of models for storage. For example, the algebra modeling storage may involve cache and/or disc memories[1]. Another criticism of classical denotational semantics is that it can be very difficult to prove properties of programs in this framework. We will see that proofs in the ADS framework can be surprisingly easy.

So-called *weakest precondition semantics* is a well-known variety of axiomatic semantics developed by Edsger Dijkstra [2, 3]. Standard textbooks using this approach have been written by David Gries [14] and Roland Backhouse [1]. In this approach, the semantics of programming language constructs is given by axioms which prescribe how those constructs transform predicates on states; thus, this semantics also has a denotational aspect, in that programs denote functions from sets of states to sets of states. A difficulty with weakest preconditions is that the semantics of iteration is rather complicated: in fact, Dijkstra, Gries and others assume that first order logic is adequate, but it seems that *infinitary logic* (i.e., the logic of infinitely long sentences!) is needed for the weakest preconditions of general iterations (see Engeler [4]); alternatively, one might use second order logic, or abandon predicates in favour of sets defined by infinite least upper bounds. In contrast, first order equational logic is sufficient for ADS.

Iteration also raises the issues of termination and well-defined values. In Gries's [14] approach to weakest preconditions, a 3-valued logic is used to handle non-termination (VDM [17] also uses a 3-valued logic); however, ADS uses only ordinary 2-valued logic. We believe that ADS gives a simpler treatment of iteration because it distinguishes between the semantics of programs and other properties of programs, such as termination and correctness. The semantics of iteration is given by two very simple OBJ equations that describe the effects on states in possible models of storage; moreover, because OBJ itself has a simple, precise semantics in equational logic, we can use the semantics of OBJ to reason about properties of such models, and hence about properties of programs. The result is that ADS provides a simple semantics, unclouded by issues of correctness or termination of programs; and in addition, we are able to use equational logic for a separate, simple treatment of correctness and termination [9]. The use of order sorted algebra plays a key rôle in our treatment of non-terminating programs; see [9] for details.

Another issue raised by various axiomatic approaches to the semantics of imperative programs, including weakest preconditions and Hoare *et al.*'s "Laws of Programming" [16], is non-determinism. Although programming languages are necessarily deterministic, a "non-deterministic" algorithm can be viewed as an equivalence class of procedures, where each procedure is a refinement, or possible implementation, of the algorithm; this is what we call loose semantics. Alternatively, non-determinism can be seen as a property of specifications in a specification language with a formal notion of one specification being more deterministic than another, as in [16]. For simplicity, we avoid both these senses of non-determinism and give a deterministic treatment of deterministic languages. However, ADS can be enriched with additional operations to support a loose semantics for non-determinism, as shown in [18].

The main motivations for non-deterministic specification are that it allows postponing design decisions, and that it leads to simple proofs of program correctness. There are even claims that, with the right semantics, algorithm development becomes an easy, near-mechanical process of looking for simple proofs. While we agree that a clear and simple formalism should lead to clarity of proof, we steer clear of any claims about the easy automatic development of algorithms; actually, there are good reasons to believe the contrary. However, in simpler cases, one can use proof construction as an aid to program construction, and we think it may be easier to do this in ADS than in more complex settings.

Our approach to properties of programs is traditional, in that we use first order logic for preconditions, postconditions, and invariants. This raises the issue of how to handle first order sentences

[1] Meseguer's so-called "logical semantics" for the λ-calculus [21] is abstract in a similar way.

in the context of an algebraic semantics. Our approach is to use standard rules of deduction to translate first order sentences into sequences of declarations and reductions[2]. Predicates are represented as `Bool`-valued operations. Assumptions and goals must be treated differently: assumptions become declarations, while goals become reductions (generally in the context of some specifically constructed declarations). For example, an assumption that is a conjunction of sentences is represented by giving one declaration for each sentence. Universal quantifiers in goals are handled by introducing new constants (this is valid by the so-called "theorem of constants"). Existential quantifiers in assumptions are handled by introducing so-called Skolem functions. The result of all this is what we call an OBJ *proof score*, that is, an OBJ program such that if its reductions all produce `true` when expected, then the desired theorem has in fact been proved.

A particular pitfall for those who wish to develop a formal semantics for imperative programming lies in the surprising variety of different kinds of variable that are involved. In fact, one must distinguish among at least the following:

1. variables that occur in programs (such as `X` in `X := X + 1`);
2. variables that range over data types used in programs (such as integers and Booleans);
3. variables that range over programs and program parts (such as expressions); and
4. variables that range over meta-logical entities used in reasoning about programs (such as computation sequences).

Furthermore, each kind of variable may have many different types. In general, textbooks on program semantics have not been very rigorous about all this. An exemplary exception is the book by John Reynolds [23]. However, such rigour is necessary for our use of OBJ as a meta-language, and also for any discussion of the semantics of procedures with parameters. The formal parameters to a procedure are variables that range over program variables; if this point is not clearly made in the semantics of procedure calls, much confusion can arise.

In summary, our Algebraic Denotational Semantics combines aspects of denotational, axiomatic and operational semantics. The denotational aspect arises because everything we specify has a denotation in an algebra; the axiomatic aspect arises from the fact that we specify these algebras using equations; and the operational aspect arises from the fact that we can symbolically execute programs using the term rewriting facility of OBJ.

3 An Example of Algebraic Denotational Semantics

Because the semantics is presented in the algebraic specification language OBJ, the first two or three weeks of the course necessarily include an introduction to that language and its semantics. We have adopted the policy of throwing students in at the deep end: in the first lecture, after explaining the goals of the course and giving a brief introduction to formal methods and semantics, we present a simple language of while programs, its semantics, and a correctness proof of a short program; the OBJ code for this example, which fits onto a handout of just three pages, is given below. To give the reader an idea of the issues raised and how the material is taught, we also include brief explanations of the most important points in the OBJ specification. These explanations are along the lines of what we actually say in the lecture. We stress that students are not expected to understand all the details of this example; the goal is simply to give the flavour of the rest of the course. We find that students manage very well to follow the gist of the example, which provides a

[2]Although it is not discussed in the course, it is possible to mechanise this translation using OBJ itself; see [6].

good opportunity to illustrate important concepts such as syntax, semantics, and loop invariants, which all play a key rôle later in the course.

The example divides naturally into three parts. The first part specifies the data types that are manipulated by the program, in this case the natural numbers. The second part specifies a theory of abstract computing machines that we call 'stores', and the syntax and semantics of while programs. The semantics is given by equational axioms that specify the effect of programs on stores. Finally, the third part defines a particular program to compute exponentiation and proves that this program is correct. The correctness proof uses the notion of loop invariant; a particular invariant for the exponentiation program is proposed, and then OBJ is used to prove that it is indeed an invariant of the program. Once the invariant is given, the correctness proof is performed entirely mechanically in OBJ: in fact, the correctness proof is *executed* in OBJ. The output produced during this execution is shown in Appendix A.

The specification of the natural numbers with some basic operations requires little explanation. The students have already had some experience with declarative definitions of data structures in a course on functional programming. The basic data structure is very simply set out in the following OBJ module:

```
obj NAT is
  sort Nat .
  op 0 : -> Nat .
  op s_ : Nat -> Nat [prec 1] .
endo
```

The chief issue here is simply to introduce OBJ's notation for declaring data structures: these modules begin and end with the keywords `obj` and `endo`; names for sorts (or data types, if you prefer) are introduced after the keyword `sort`; and names for operations are introduced after the keyword `op`. We also stress, without going into details, that OBJ's denotational semantics gives a precise meaning to such declarations; in this case, it so happens that the denotational semantics gives us the 'obvious' denotation, i.e., the natural numbers. Some operations on numbers are specified in the next OBJ module:

```
obj NATOPS is pr NAT .
  op 1 : -> Nat .
  eq 1 = s 0 .
  op _+_ : Nat Nat -> Nat [assoc comm prec 3] .
  vars M N : Nat .
  eq  M + 0  =  M .
  eq  M + s N  =  s(M + N) .
  op _*_ : Nat Nat -> Nat [assoc comm prec 2] .
  eq  M * 0  =  0 .
  eq  M * s N  =  M * N + M .
  op _**_ : Nat Nat -> Nat [prec 4] .
  eq  M ** 0  =  1 .
  eq  M ** s N  =  (M ** N) * M .
  op _-_ : Nat Nat -> Nat .
  eq  M - 0  =  M .
  eq  0 - M  =  0 .
  eq  s M - s N  =  M - N .
```

```
   op _%2 : Nat -> Nat [prec 10] .
   eq  0 %2  =  0 .
   eq  s 0 %2  =  0 .
   eq  s s N %2  =  s(N %2) .
   op even_ : Nat -> Bool  [prec 10] .
   eq  even 0  =  true .
   eq  even s 0  =  false .
   eq  even s s M  =  even M .
   op pos_ : Nat -> Bool .
   eq  pos 0  =  false .
   eq  pos s N  =  true .
   [lemma1] cq (N * N)**(M %2) = N ** M if even M .
   [lemma2] cq N *(N **(M - s 0)) = N ** M if pos M .
 endo
```

Again, this requires little explanation. Modules are imported using the keyword `pr` (for 'protecting', in the sense that the declarations in the module do not add any new numbers to the specification in `NAT`, nor do they identify distinct numbers). The operation `%2` is integer division by two. We point out that in OBJ's denotational semantics, equational axioms, introduced by the keyword `eq` (or `cq` for conditional equations), assert equalities, while in OBJ's operational semantics, they are used left-to-right to rewrite terms. For example,

```
((s 0)+(s 0))**(s s s 0)
```

is rewritten in the obvious way to

```
s s s s s s s s 0 .
```

The two labelled equations at the end of the `NATOPS` module are lemmas that are needed for executing the correctness proof of the exponentiation program in OBJ. They were obtained by looking at OBJ's output when it failed to complete the proof: instead of returning the result `true`, it returned a term containing instances of the lefthand sides of the given lemmas. This enabled us to see that these particular additional properties were needed[3]. In a similar way, the fact that OBJ returns subexpressions that it cannot yet reduce is of considerable help with the task of strengthening invariants that is so often necessary in construcing correctness proofs of imperative programs.

The real business of Algebraic Denotational Semantics begins with the next module, `STORE`, which defines a class of abstract machines that associate natural number values with variables in the programming language. For these variables, we use OBJ's built in sort `Id` of 'quoted identifiers'. The declaration

```
pr QID *(sort Id to Var) .
```

imports this built in sort and renames it to `Var`. We stress that program variables of sort `Var` are very different from the variables used in OBJ equations (these are introduced after the keyword `var`, with a small 'v').

[3]These properties can also be proved using OBJ, since inductive proofs of properties of data types are very similar to proofs of invariants of programs. This is not emphasised in our course, but it is discussed in [6].

```
th STORE is pr NATOPS .
            pr QID *(sort Id to Var) .
  sort Store .
  op initial : -> Store .
  op _[[_]] : Store Var -> Nat .
  op (_;_:_) : Store Var Nat -> Store .
  vars X Y : Var .  var  S : Store .  var  N : Nat .
  eq  initial [[X]]  =  0 .
  eq  S ; X : N [[X]]  =  N .
  cq  S ; X : N [[Y]]  =  S[[Y]] if X =/= Y .
endth
```

This module begins and ends with the OBJ keywords `th` and `endth`, which indicates that the module defines a *theory* of abstract machines: anything that associates values with variables in the way described by the equations is acceptable as a model of this theory, whereas the previous modules, which begin and end with the keywords `obj` and `endo`, are intended to denote the 'obvious' or 'standard' (i.e., technically, the initial [10]) model; that is, they are intended to specify the data type of natural numbers. A `Store` is therefore anything which has a distinguished constant `initial`, an operation

```
_[[_]] : Store Var -> Nat
```

(which allows examining the value a store associates to a given program variable), and an operation

```
(_;_:_) : Store Var Nat -> Store
```

(which updates the value associated with a given program variable) satisfying the equations given in the module. In a similar way, a theory of monoids could be given by declaring a sort with a constant and a binary operation which satisfy identity and associativity equations. The semantics of imperative programs will be given by equations that specify the effect of programs on stores.

The syntax of our programming language is defined simply by declaring the constructs of the language as operations. The module `EXP` defines integer and boolean expressions for a simple programming language. The equations of the module give the semantics of these expressions, relative to a given `Store`, in the following:

```
obj EXP is pr STORE .
  sorts Exp Bexp .
  subsorts Nat Var < Exp .
  subsorts Bool < Bexp .
  op _*_ : Exp Exp -> Exp .
  op _-_ : Exp Exp -> Exp .
  op _%2 : Exp -> Exp .
  op even_ : Exp -> Bexp .
  op pos_ : Exp -> Bexp .
  op _[[_]] : Store Exp -> Nat .
  op _[[_]] : Store Bexp -> Bool .
  var  S : Store .  var  N : Nat .  var  B : Bool .
  vars E E' : Exp .  var  BX : Bexp .
  eq  S[[N]]  =  N .
  eq  S[[B]]  =  B .
```

```
  eq  S[[E + E']]  =  S[[E]] + S[[E']] .
  eq  S[[E * E']]  =  S[[E]] * S[[E']] .
  eq  S[[E - E']]  =  S[[E]] - S[[E']] .
  eq  S[[E  %2]]   =  S[[E]] %2 .
  eq  S[[even E]]  =  even(S[[E]]) .
  eq  S[[pos E]]   =  pos(S[[E]]) .
endo
```

The key point is that expressions in the language 'look like' natural numbers, except that they may contain variables; if we can assign natural number values to these variables (by means of a `Store`), then we can assign natural number values to the expressions which contain those variables. The equations in this module demonstrate the difference between syntax and semantics: for example, the operation `_+_`, which takes two expressions as arguments and returns an expression as result, is purely syntactic, but the equations which describe semantics of expressions relative to a given `Store` state that it is *interpreted* as addition on the semantic domain of natural numbers.

Similarly, the module `PGM` defines the syntax and semantics of a simple programming language. As in `EXP`, the operations declared in the module specify the syntax of the language, and the equations specify its semantics.

```
obj PGM is pr EXP .
           pr STORE .
  sort Pgm .
  op _:=_ : Var Exp -> Pgm .
  op _;_ : Pgm Pgm -> Pgm [assoc prec 50] .
  op if_then_else_fi : Bexp Pgm Pgm -> Pgm .
  op while_do_od : Bexp Pgm -> Pgm .
  op _;_ : Store Pgm -> Store .
  var  S : Store .  var  X : Var . var  E : Exp .
  var  BX : Bexp .  vars P P' : Pgm .
  eq  S ; (X := E)  =  S ; X : S[[E]] .
  eq  S ; (P ; P') =  (S ; P); P' .
  cq  S ; if BX then P else P' fi  =  S ; P
      if  S[[BX]] .
  cq  S ; if BX then P else P' fi  =  S ; P'
      if  not(S[[BX]]) .
  cq  S ; while BX do P od  =  S ; P ; while BX do P od
      if  S[[BX]] .
  cq  S ; while BX do P od  =  S
      if  not(S[[BX]]) .
endo
```

The first four operations declared in the module give the constructs of our simple programming language: assignment, sequential composition, conditional, and iteration. The fifth operation,

```
_;_ : Store Pgm -> Store ,
```

allows evaluating a given program in a given store; the modified store is returned as result. It is by means of this operation that we define the semantics of the language, by specifying the effect of each construct on a given store. The first equation in this module defines the semantics of assignment, the programming construct that underlies all imperative programming languages. The equation states that the effect of an assignment on a store is to update the value associated with the given variable. The remaining equations specify the effect of the other constructs in the programming language on stores.

Note that it is possible to specify in a very short time a semantic domain (the natural numbers), a theory of abstract machines, the syntax of a programming language, and the semantics of that language. Moreover, while the specification has an abstract algebraic denotation, the operational semantics of OBJ provides an operational semantics for the example programming language! OBJ evaluates terms by rewriting them according to the given equations, from left to right; the equations in our specification allow OBJ to evaluate programs written in the example programming language. For example, the module `POW` defines a program that computes exponentiation:

```
th POW is inc PGM .
  ops a0 b0 : -> Nat .
  let init  = 'A := a0 ; 'B := b0 ; 'C := 1 .
  let step  = 'B := 'B - 1 ; 'C := 'C * 'A .
  let inner = 'A := 'A * 'A ; 'B := 'B %2 .
  let outer = while even 'B do inner od ; step .
  let pow   = init ; while pos 'B do outer od .
endth
```

Here, the operations `a0` and `b0` are used to represent the initial values[4] given to the program, which sets the variable `'C` to `a0` to the power of `b0`. The keyword `let` introduces an abbreviation. Such abbreviations are treated by OBJ in the same way as equations, so the command

```
red pow .
```

causes OBJ to reduce the term `pow` using the given equations, giving the following result:

```
'A := a0 ; 'B := b0 ; 'C := 1 ;
while pos 'B
do while even 'B
   do 'A := 'A * 'A ; 'B := 'B %2 od ;
   'B := 'B - 1 ; 'C := 'C * 'A
od
```

The equations defining the semantics of the programming language can also be used to evaluate the results of running the program `pow`. For example,

```
open POW .
eq a0 = s s 0 .
eq b0 = s s s 0 .
red initial ; pow [['C]] .  ***> should be: 8
close
```

sets the initial values of `a0` and `b0` to `2` and `3`, respectively, and asks for the value of the variable `'C` after running the program in the initial state (in which all variables have the value `0`). The result returned is, as it should be, `8` (of course, in the form of 8 `s`'s followed by a `0`). The command `***>`

[4]Such variables are sometimes called "ghost variables".

indicates that what follows it (on the same line) is a comment that should be printed in the output when it is executed.

The remainder of the OBJ code presents a proof that the program is correct. The proof uses the notion of invariant, which is central to the course, and can be explained at a very intuitive level by drawing a flow chart to give a graphic representation of the nested loop structure[5]. An invariant of the program is a property that holds before the loop is entered, and is preserved by each iteration of both the inner and outer loop, and it therefore holds when (or if!) the loop is exited. An appropriate invariant for the correctness proof of `POW` is given in the following:

```
obj PROOF is pr POW .
  op inv : Store -> Bool .
  var S : Store .
  eq  inv(S)  =  ((S[['A]]) ** (S[['B]])) * (S[['C]]) == a0 ** b0 .
  op s : -> Store .
  ops a b c : -> Nat .
  eq s[['A]] = a .
  eq s[['B]] = b .
  eq s[['C]] = c .
endo
```

As well as defining the invariant for the proof, this module introduces an arbitrary store `s`, and abbreviations `a`, `b` and `c` for the values of `'A`, `'B` and `'C` in this store. The program is proved correct by showing that `init` establishes the invariant, and that the truth of the invariant is preserved by the bodies of the inner and outer loops. Each part of the proof is achieved by an OBJ reduction. The reduction

```
open PROOF .
red inv(s ; init) .  ***> should be: true
close
```

shows that the invariant holds initially before the loop is entered, and the two reductions below show that if the invariant holds before the body of the loop is evaluated, then it also holds after the body is evaluated. The assumption that the invariant holds before the body is evaluated is made by asserting the equations that precede the final two reductions.

```
open PROOF .
eq (a ** b)* c = a0 ** b0 .
eq pos b = true .
eq even b = true .
red inv(s ; inner).  ***> should be: true
close

open PROOF .
eq (a ** b)* c = a0 ** b0 .
eq even b = false .  ***> therefore by [lemma3]:
eq pos b = true .
red inv(s ; step) .  ***> should be: true
close
```

[5]Indeed, this is how it was historically introduced (independently) by Turing and von Neumann.

The program `pow` terminates when the guard of the loop fails, that is, when `'B` finally becomes zero, assuming the precondition that the value of `'B` is initially non-negative. In this case, the invariant still holds, and it says that `'C` has the value `a0 ** b0`, as desired. (Later in the course, this method is stated as a formal theorem.) This proof of partial correctness extends to a proof of total correctness by showing that the program actually terminates, and this is done in our course and in our book, again using simple reductions that are performed in OBJ.

It is worth pausing to notice that all we had to do was set up the proof and let OBJ do all the tedious computations, and OBJ even provided hints for the non-routine parts, including the lemmas that were needed. The entire example took only three pages of OBJ code, including defining the numbers, giving the syntax and semantics of a simple imperative programming language, and using that semantics in a correctness proof for an exponentiation program. Moreover, both the semantics of the language and the correctness proof are executable.

The course and the book that developed from it [9] give a more encompassing semantics for numbers, as well as for arrays and procedures, and of course, there are many more examples using these and other features. Termination and total correctness proofs are also treated.

4 Discussion and Conclusions

The example of Algebraic Denotational Semantics given in the previous section illustrates the advantages of using OBJ to express the semantics of imperative programs. The example is sufficiently short, intuitive and readable to be used in an introductory lecture, yet because OBJ is a formal, implemented notation, an extremely high level of formality and mathematical rigour is achieved. OBJ's algebraic semantics gives a precise meaning to the specification, while OBJ's operational semantics gives a way to execute programs and proofs. The operational component allows students to gain insight into the constructs of our programming language, and the nature of correctness proofs. This course is, we believe, unique in Computing Science education in that it is *executable*. Its success arises in large measure from the simplicity and efficiency of equational logic, with its armoury of powerful theorems and algorithms.

Contrary to claims in many books on formal methods, we believe that programming is not purely formal. Programming is a skill, and to be really good at it can take years of hard work. One must learn how to use tools like configuration managers and debuggers. For large programs, one must learn how to work in a team; and for really large projects, management and other social issues are often dominant [7]. One must learn discipline and organisation, and how to read and write documentation. One must keep learning new languages, tools, concepts, algorithms, and skills; sometimes one must even invent these things. Software Engineering is a difficult area requiring diverse skills and knowledge, and the material taught in our course provides little help with those aspects that are not formal and cannot be formalized.

It is unhealthy to confuse a formal notation with a formal method. A *method* should say how to do something, whereas a *notation* allows one to say something [7]. Thus, OBJ is only a notation, but using it as described in our course and book gives a method for proving properties of imperative programs. This method is not completely mechanical, because some of the problems involved are uncomputable (such as finding loop invariants), although of course, the OBJ3 implementation[6] is completely mechanical. Nevertheless, the method is surprisingly effective, in part because OBJ does all the routine work, and even provides hints to help with doing much of the non-routine work.

[6] "OBJ" refers to the general design, while "OBJ3" refers to a specific implementation.

The problems that arise for large programs are qualitatively very different from those that arise for small programs. The student should not be led into believing that it is just as easy to find specifications and invariants for the flight control software of a real aircraft as it is for a sorting algorithm, or indeed that finding specifications and invariants will be a major activity in real industrial work. On the contrary, it turns out that finding requirements (i.e., determining what kind of system to build), structuring the system (modular design), understanding what has already been done (reading documentation and talking to others), and organizing the efforts of a large team, are all much more important for a large system development effort.

Nevertheless, we hope that having a precise understanding of programming constructs and of what programs mean will be a good basis for further professional development. In particular, we hope that the way we use OBJ for specification can be an inspiration for documentation, and that the way we use OBJ modules can be an inspiration for structuring large programs. We also hope that the material covered in the course will be useful to those who wish to design new languages, new computers, new operating systems, etc., or to develop new theories and formal methods that support such endeavours in Software Engineering. We believe that the sort of algebraic approach developed in the course is particularly promising for such efforts, and we hope to provide a foundation for approaching the large literature that applies algebraic techniques in Computing Science in general and to formal methods in particular.

A Executing the Proof Score

In a certain sense, not only our course, but also this paper is executable! OBJ3 has a facility that extracts OBJ code from an arbitrary text file and then executes it. The (unedited) result of using this facility on the LaTeX source file for this paper is given below. It shows OBJ3 executing the correctness proof given in the body of this paper, and outputting `true` for the initialization and loop subproofs of invariance, as expected.

```
                     \|||||||||||||||||/
                   --- Welcome to OBJ3 ---
                     /|||||||||||||||||\
         OBJ3 version 2.02 built: 1992 May 18 Mon 9:57:30
            Copyright 1988,1989,1991 SRI International
                   1995 Nov 5 Sun 16:59:11
OBJ>
==========================================
obj NAT
Warning: redefining module NAT
==========================================
obj NATOPS
==========================================
th STORE
==========================================
obj EXP
==========================================
obj PGM
==========================================
th POW
```

```
============================================
reduce in POW : pow
rewrites: 7
result Pgm: 'A := a0 ; 'B := b0 ; 'C := s 0 ; while pos 'B do while
    even 'B do 'A := ('A * 'A) ; 'B := ('B %2) od ; 'B := ('B - s 0) ;
    'C := ('C * 'A) od
============================================
open POW
============================================
eq a0 = s s 0 .
============================================
eq b0 = s s s 0 .
============================================
reduce in POW : (initial ; pow)[['C]]
rewrites: 223
result Nat: s (s (s (s (s (s (s (s 0))))))))
============================================
***> should be: 8
============================================
close
============================================
obj PROOF
============================================
open PROOF
============================================
reduce in PROOF : inv(s ; init)
rewrites: 27
result Bool: true
============================================
***> should be: true
============================================
close
============================================
open PROOF
============================================
eq ( a ** b ) * c = a0 ** b0 .
============================================
eq pos b = true .
============================================
eq even b = true .
============================================
reduce in PROOF : inv(s ; inner)
rewrites: 27
result Bool: true
============================================
***> should be: true
============================================
close
```

```
===========================================
open PROOF
===========================================
eq ( a ** b ) * c = a0 ** b0 .
===========================================
eq even b = false .
===========================================
***> therefore by [lemma3]:
===========================================
eq pos b = true .
===========================================
reduce in PROOF : inv(s ; step)
rewrites: 31
result Bool: true
===========================================
***> should be: true
===========================================
close
OBJ> Bye.
```

References

[1] Roland Backhouse. *Program Construction and Verification.* Prentice-Hall, 1986.

[2] Edsger Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the Association for Computing Machinery*, 18:453–457, 1975.

[3] Edsger Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[4] Erwin Engeler. Structure and meaning of elementary programs. In Erwin Engeler, editor, *Symposium on Semantics of Algorithmic Languages*, pages 89–101. Springer, 1971. Lecture Notes in Mathematics, Volume 188.

[5] Robert Floyd. Assigning meanings to programs. In Jacob Schwartz, editor, *Proceedings, Symposia Applied Mathematics*, volume 19, pages 19–32. American Mathematical Society, 1967.

[6] Joseph Goguen. *Theorem Proving and Algebra.* MIT, to appear 1996.

[7] Joseph Goguen and Luqi. Formal methods and social context in software development. In Peter Mosses, Mogens Nielsen, and Michael Schwartzbach, editors, *Proceedings, Sixth International Joint Conference on Theory and Practice of Software Development*, pages 62–81. Springer-Verlag Lecture Notes in Computer Science 915, 1995.

[8] Joseph Goguen and Grant Malcolm. Proof of correctness of object representations. In A. W. Roscoe, editor, *A Classical Mind: essays dedicated to C.A.R. Hoare*, chapter 8, pages 119–142. Prentice-Hall, 1994.

[9] Joseph Goguen and Grant Malcolm. *Algebraic Semantics of Imperative Programs.* MIT Press, to appear 1996.

[10] Joseph Goguen, James Thatcher, and Eric Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. Technical Report RC 6487, IBM T.J. Watson Research Center, October 1976. In *Current Trends in Programming Methodology, IV*, Raymond Yeh, editor, Prentice-Hall, 1978, pages 80–149.

[11] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Case Studies in Algebraic Specification Using OBJ*. Cambridge University Press, to appear 1996. Also available as a technical report from SRI International.

[12] Herman Goldstine and John von Neumann. Planning and coding of problems for an electronic computing instrument. In A. Traub, editor, *Collection Works of J. von Neumann*, pages 80–151. Pergamon, 1949. Originally, a report of the U.S. Ordinance Department.

[13] Michael J.C. Gordon. *The Denotational Description of Programming Languages*. Springer, 1979.

[14] David Gries. *The Science of Programming*. Springer, 1981.

[15] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the Association for Computing Machinery*, 12(10):576–580, October 1969.

[16] C.A.R. Hoare, J. Michael Spivey, Ian Hayes, Jifeng He, Carroll Morgan, A. William Roscoe, Jeffrey Saunders, Ib Sorenson, and Bernard Sufrin. Laws of programming. *Communications of the Association for Computing Machinery*, 30(8):672–686, 1987.

[17] Cliff B. Jones. *Systematic Software Development using VDM*. Prentice-Hall, 1986.

[18] Grant Malcolm and Joseph Goguen. Algebraic semantics of nondeterministic choice. Technical Report PRG-TR-5-95, Programming Research Group, Oxford University, 1995.

[19] Grant Malcolm and Joseph Goguen. Proving correctness of refinement and implementation. Technical Monograph PRG-114, Programming Research Group, Oxford University, 1994.

[20] John McCarthy. Towards a mathematical science of computation. In *Information Processing '62*, pages 21–28. North-Holland, 1962. Proceedings of 1962 IFIP Congress.

[21] José Meseguer. Conditional rewriting logic: Deduction, models and concurrency. In Stéphane Kaplan and Misuhiro Okada, editors, *Conditional and Typed Rewriting Systems*, pages 64–91. Springer, 1991. Lecture Notes in Computer Science, Volume 516.

[22] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN–19, Computer Science Department, Aarhus University, September 1981.

[23] John C. Reynolds. *The Craft of Programming Languages*. Prentice-Hall, 1981.

[24] Dana Scott and Christopher Strachey. Towards a mathematical semantics for computer languages. In *Proceedings, 21st Symposium on Computers and Automata*, pages 19–46. Polytechnic Institute of Brooklyn, 1971. Also Programming Research Group Technical Monograph PRG–6, Oxford University.

[25] Joseph Stoy. *Denotational Semantics of Programming Languages: The Scott-Strachey Approach to Programming Language Theory*. MIT, 1977.

[26] Alan M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. University Mathematics Lab, Cambridge University, 1949.