Teaching Semantics of Programming Languages with Modular SOS

Peter D. Mosses
Department of Computer Science, Swansea University
Singleton Park, Swansea, SA2 8PP, UK
www.cs.swan.ac.uk/~cspdm/
p.d.mosses@swan.ac.uk

Abstract

Most undergraduate courses on formal semantics are based on conventional Structural Operational Semantics (SOS) and/or Denotational Semantics. Typically, they give semantic descriptions of a series of small programming languages, starting from a very simple one, and subsequently extending it with various new features. For each extension, however, it is usually necessary to revisit the description of the constructs of the initial language, and reformulate it to take account of the new features – although the required reformulation is often so routine that it may be left as an exercise.

In contrast, a course based on *Modular SOS* gives an independent description of each language construct, and no reformulation at all is needed when adding new features. Such a course has been given for 3rd-year undergraduates at the University of Aarhus, Denmark, 2001–2004. A further novelty of the course was to use substantial fragments of a real programming language (Standard ML) for illustration and exercises. Moreover, tool support for Modular SOS enabled the students to add their own descriptions of new constructs to a given language description, and to check empirically whether the resulting semantics of programs in the extended language was as intended.

Keywords: Teaching, Formal Semantics, Programming Languages, Modular SOS

1. INTRODUCTION

Each year from 2001 to 2004, the author gave a 3rd-year course on *Semantics of Programming Languages* at the University of Aarhus, Denmark. The course covered both the conceptual analysis and the formal semantic description of programming languages: it explained how to analyse programming constructs in terms of fundamental concepts, and how to formulate precise descriptions of their intended interpretation using a structural approach to operational semantics. The approach was illustrated mainly by analysing and describing constructs taken from the functional programming language Standard ML; familiarity with Standard ML (or a related language) was assumed. The analysis and description of constructs from other languages was explored in the exercises.

An important novel aspect of the material was the use of *Modular SOS* (MSOS) [5], a recently-developed variant of the well-known *Structural Operational Semantics* (SOS) framework [7]. As its name suggests, this variant has the advantage of providing an exceptionally high degree of modularity: individual language constructs can be described independently, and freely combined to obtain descriptions of complete languages. The descriptions of the individual constructs remain unchanged when further constructs are added to the described language. Conventional SOS descriptions do not enjoy such modularity. For instance, an SOS description of constructs from a pure functional language requires extensive reformulation when expressions are extended to allow side-effects or concurrent processes. Using MSOS, the description of purely functional constructs does not depend on whether expression evaluation might involve side effects or other processes.

Descriptions in MSOS could be automatically translated into programs in the logic programming language Prolog, using software provided to the students. This provided the basis for prototype implementations of languages involving the described constructs: test programs could be run according to their specified semantics. The main purpose of such prototyping was to check that the specified semantics conformed to expectations; inspecting the course of program executions in the prototype implementation also helped to understand the techniques used in MSOS.

The course also considered (albeit very briefly) alternative approaches to formal semantics, including action semantics, axiomatic semantics, denotational semantics, and further variants of operational semantics. The aim was merely to provide students with a general impression of previous approaches to formal semantics; references to the literature were provided for those who might wish to continue with more advanced studies in this area.

In its final form in 2004, the course on semantics took seven weeks, with three 45-minute lectures and three 45-minute exercise classes each week. The examination for the course required each student to complete a small project over a couple of days. About 110 students took the course, and the pass rate was more than 75%. (From 2001 to 2003, the same topics had been covered as part of a combined 15-week course on *Programming Languages and Formal Semantics*, with two substantial coursework assignments and a 4-hour open-book written examination.) The feedback from the students was generally positive regarding the lectures, the tool support, and the notes.

Most of the students took the course in their 5th semester; they were assumed to have previously completed an introductory Programming course (the one at Aarhus included programming in Java), a course on Models and Logic, and a course on Programming Languages (the one at Aarhus covered functional programming in ML and, to a lesser extent, logic programming in Prolog).

The rest of this paper illustrates the reported approach to teaching formal semantics using Modular SOS by giving extracts from Section 1.2 of the (unpublished) lecture notes used for the course. The course web pages from 2004 are still available¹, and provide links not only to the full lecture notes but also to the accompanying software. The lecture notes and accompanying software were also used in 2004 by Bernd Meyer and Maria Garcia de la Banda at the School of Computer Science and Software Engineering, Monash University, Australia, as part of their course on *Programming Language Concepts and Semantics*.

2. FORMAL LANGUAGE DESCRIPTIONS

The following extracts from section 1.2 of the lecture notes attempt to give a fair impression of the approach used. For brevity,² explanations of concepts assumed to be familiar to the BCS-FACS meeting participants are generally omitted, and only representative excerpts of the illustrative examples are given. These examples are all based on a small imperative language called *bc* [2]; subsequent chapters of the lecture notes use constructs taken from Standard ML for illustration.

2.1. Syntax, Semantics, and Formality

The operational framework known as Structural Operational Semantics (SOS) [7] is a good compromise between simplicity and practical applicability, and it has been widely taught at the undergraduate level [3, 6, 8, 9]. The modular variant of SOS used in the present notes, called MSOS [5], has some significant pragmatic advantages, but otherwise remains conceptually very close to the original SOS framework.

2.2. Concrete Syntax

Here, we shall adopt one particular technique that generalizes BNF, called *Definite Clause Grammars* (DCGs) [1, Section 7]. These grammars were developed for use in describing natural

¹<http://www.daimi.au.dk/jwig-cnn/dSem/>

²The full version of the corresponding section in the lecture notes is 21 pages.

languages, but they turn out to be rather well-suited for describing programming languages as well. Prolog supports parsing for *arbitrary* DCGs (although left-recursive rules can lead to nontermination). Table 1 shows an excerpt from a DCG for simple and structured statements in *bc*. It is straightforward to understand each rule of the form ‘ $\text{---}\rightarrow\text{---}$.’ as a production of a context-free grammar: the lowercase words are nonterminal symbols, the characters in double quotation marks form terminal symbols, and the commas simply form sequences of symbols. The nonterminal symbol ‘*i*’ is defined to match arbitrary horizontal space and comments.

TABLE 1: Concrete syntax for a subset of *bc* (*extract*)

```

3  stmts --> stmt, i, sep, optnewlines, stmts.
4  stmts --> stmt.
5
6  sep --> ";".
7  sep --> newline.
8
9  stmt --> assign_expr.
10 stmt --> normal_expr.
11 stmt --> "{", optnewlines, stmts, optnewlines, "}".
12 stmt --> "{", optnewlines, "}".
13 stmt --> "if", i, "(", i, expr, i, ")", i, stmt.

```

2.3. Abstract Syntax

Table 2 specifies an abstract syntax which represents the deep structure of the commands (i.e., statements) and expressions whose concrete syntax was given in Table 1.

TABLE 2: Abstract syntax for a subset of *bc* (*extract*)

```

1  C:  Cmd ::= skip | seq(Cmd,Cmd) | cond-nz(Exp,Cmd)
2        | while-nz(Exp,Cmd) | effect(Exp) | print(Exp)
3
4  E:  Exp ::= Boolean | Integer | Var
5        | assign-seq(Var,Exp) | app(Op,Arg)

```

2.4. Mapping Concrete to Abstract Syntax

To obtain the semantics of a concrete program, we need to map it (unambiguously) to the corresponding abstract syntax tree. Such a mapping can be specified in various ways, e.g., as a function from concrete derivation trees (produced by parsing the program text) to abstract syntax trees. DCGs (in contrast to BNF) allow the mapping to abstract syntax trees to be specified along with the concrete syntax. Table 3 shows a DCG which augments the concrete syntax from Table 1 with a mapping to the abstract syntax of Table 2.

TABLE 3: Mapping concrete to abstract syntax for a subset of *bc* (*extract*)

```

3  stmts(seq(C1,C2)) --> stmt(C1), i, sep, optnewlines, stmts(C2).
4  stmts(C) --> stmt(C).
5  sep --> ";".
6  sep --> newline.
7  stmt(effect(E)) --> assign_expr(E).
8  stmt(print(E)) --> normal_expr(E).
9  stmt(C) --> "{", optnewlines, stmts(C), optnewlines, "}".
10 stmt(skip) --> "{", optnewlines, "}".
11 stmt(cond_nz(E,C)) -->
12   "if", i, "(", i, expr(E), i, ")", i, stmt(C).

```

2.5. Static Semantics

Table 4 gives an impression of how static semantics can be specified in MSOS. The notation $E \implies VT$ asserts that expression E has values of type VT (similarly for commands, etc.), and a horizontal line indicates that the assertion below the line holds whenever all the assertions above the line hold.³

TABLE 4: Static semantics for a subset of bc (*extract*)

```

21  /* Command typechecking rules: */
22
23  skip:Cmd ==> void
24
25  C1 ==> void, C2 ==> void
26  -----
27  seq(C1,C2):Cmd ==> void
28
29  E ==> int, C ==> void
30  -----
31  cond-nz(E,C):Cmd ==> void

```

2.6. Dynamic Semantics

Table 5 gives an impression of how dynamic semantics is specified in MSOS. The notation used is compatible with that used for describing static semantics in Table 4, but we use a different notation for transitions, to avoid confusion between static and dynamic semantics: $E \dashrightarrow V$ asserts that E evaluates to value V . However, evaluation of expressions (and execution of most other constructs in programming languages) is essentially a gradual process, often taking many small steps. This is reflected by considering intermediate states for evaluation: $E \dashrightarrow E'$ indicates that the next step of evaluating E leads to the expression E' , which need not be the value computed by E .

TABLE 5: Dynamic semantics for a subset of bc (*extract*)

```

21  Label = {env:Env,store,store':Store,out':Value*,...}
22
23
24  /* Command transition rules: */
25
26  C1 --{...}-> C1'
27  -----
28  seq(C1,C2):Cmd --{...}-> seq(C1',C2)
29
30  seq(skip,C2):Cmd ---> C2
31
32
33  E --{...}-> E'
34  -----
35  cond-nz(E,C):Cmd --{...}-> cond-nz(E',C)
36
37  cond-nz(0,C):Cmd ---> skip
38
39  N \= 0
40  -----
41  cond-nz(N,C):Cmd ---> C

```

More generally, the assertion $E \dashrightarrow\{...\} E'$ allows a step to have *side-effects*, such as assigning to variables. We refer to $\{...\}$ as the *label* of the step. The exploitation of labels to

³The fundamental difference between such an inference rule and an implication formula is significant in connection with proof of bisimulation equivalence, which is covered in Chapter 4 of the lecture notes.

represent side-effects and other kinds of information processing is one of the main characteristics of MSOS [4]. The label corresponds to a *record value*: fields such as `env` and `store` give access to information available before the step, whereas primed fields such as `store'` and `out'` are set when the step is taken. The ellipsis symbol '`...`' in a label corresponds to a variable ranging over the unmentioned fields of the record value, and similarly when specifying the set `Label` of labels; such information hiding is crucial for the modularity of MSOS.

2.7. Complete Descriptions and Modularization

To maximize reusability, each module should specify the semantics of just a *single construct*. The descriptions of the static and dynamic semantics of a construct are both based on its abstract syntax, but they are otherwise quite independent. Moreover, the abstract syntax of a construct is of independent interest. We shall therefore divide each module that describes an abstract construct into separate descriptions of abstract syntax, static semantics, and dynamic semantics, where the semantic descriptions (implicitly) refer to the abstract syntax description.

To get an impression of how this works in practice, consider the descriptions of the abstract syntax and semantics given earlier. We shall use the symbols that name sets of abstract syntax trees and their constructors to form module names. For example, `Cmd/seq/ABS` names the module that specifies the abstract syntax of the construct `seq(Cmd, Cmd)` (shown in Table 6), `Cmd/seq/CHK` its static semantics (Table 7), and `Cmd/seq/RUN` its dynamic semantics (Table 8). Auxiliary modules (not shown here) declare `C` as a variable ranging over the set `Cmd`, and specify common notation required in the static, resp. dynamic semantics of *arbitrary* commands.

TABLE 6: `Cmd/seq/ABS`: Abstract syntax of sequential commands

<code>Cmd ::= seq(Cmd, Cmd)</code>

TABLE 7: `Cmd/seq/CHK`: Static semantics of sequential commands

<code>C1 ==> void, C2 ==> void</code> <hr/> <code>seq(C1, C2) : Cmd ==> void</code>

TABLE 8: `Cmd/seq/RUN`: Dynamic semantics of sequential commands

<code>C1 --{...}-> C1'</code> <hr/> <code>seq(C1, C2) : Cmd --{...}-> seq(C1', C2)</code> <code>seq(skip, C2) : Cmd ----> C2</code>
--

Given a collection of named modules covering all the abstract constructs involved in a particular programming language, the semantic description of the language is given by listing the required modules, as illustrated in Table 9. The provided tool support for MSOS (itself implemented in Prolog) allows this module and all the referenced modules to be (separately) compiled into Prolog. Loading the Prolog code, together with a DCG that specifies a mapping from concrete syntax to abstract constructs (as illustrated in Table 3) and some fixed Prolog modules, provides an interpreter for the specified language, allowing test programs to be type-checked and run.

2.8. Practical Uses

The design and implementation of programming languages is a topic of major importance in Computer Science. Few other semantic frameworks come close to the degree of modularization provided by MSOS. This pragmatic feature of MSOS descriptions, together with proper tool

TABLE 9: Modular abstract syntax for a subset of bc

3	see	Cmd/skip,	Cmd/seq,	Cmd/cond-nz,	
4		Cmd/while-nz,	Cmd/effect,	Cmd/print	
5					
6	see	Exp/Boolean,	Exp/Integer,	Exp/Var,	
7		Exp/assign-seq,	Exp/app-Op		
8					
9	see	Var/Id, Id/id			
10					
11	see	Op/plus, Op/minus, Op/times, Op/div, Op/mod, Op/ord,			
12		Op/lt, Op/leq, Op/gt, Op/geq, Op/eq, Op/neq			
13					
14	see	Arg/Exp, Arg/tup-seq			

support for their development and prototyping, should greatly facilitate description and reasoning about full-scale programming languages. Some case studies have already been carried out.⁴

3. CONCLUSION

The course described in this paper has several novel aspects:

- use of a modular framework for semantic description, avoiding the need to reformulate descriptions of existing constructs when adding new features;
- a clear distinction between concrete constructs of particular languages and language-independent abstract constructs; and
- illustrative examples of semantics involving concrete constructs found in real programming languages (bc, Standard ML).

The provision of tool support (following [6, 8]) helps to make formal semantics more accessible to weaker students, and allows students to check their answers to exercises for themselves. The latter aspect was much appreciated by the teaching assistants, as well as by the students. The reader is referred to the lecture notes⁵ for full details of the contents of the course.

The author is grateful to the anonymous referees for their helpful suggestions.

REFERENCES

- [1] P. Blackburn, J. Bos, and K. Striegnitz. Learn Prolog now! <http://www.coli.uni-sb.de/~kris/prolog-course/html/>, 2003.
- [2] Free Software Foundation. *GNU bc Manual*, 2001. <http://www.gnu.org/software/bc/>.
- [3] M. Hennessy. *The Semantics of Programming Languages: An Elementary Introduction Using Structural Operational Semantics*. Wiley, New York, 1990.
- [4] P. D. Mosses. Exploiting labels in structural operational semantics. In H. Haddad, A. Omicini, R. L. Wainwright, and L. M. Liebrock, editors, *SAC 2004*, pages 1476–1481. ACM, 2004.
- [5] P. D. Mosses. Modular structural operational semantics. *J. Logic and Algebraic Programming*, 60–61:195–228, 2004. Special issue on SOS.
- [6] H. R. Nielson and F. Nielson. *Semantics with Applications: A Formal Introduction*. Wiley, Chichester, UK, 1992.
- [7] G. D. Plotkin. A structural approach to operational semantics. *J. Logic and Algebraic Programming*, 60–61:17–139, 2004. Originally published as DAIMI FN-19, Dept. of Computer Science, Univ. of Aarhus, 1981.
- [8] K. Slonneger and B. L. Kurtz. *Formal Syntax and Semantics of Programming Languages: A Laboratory Based Approach*. Addison-Wesley, 1995.
- [9] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, 1993.

⁴<http://maude-msos-tool.sourceforge.net/>

⁵The lecture notes are currently available from <http://www.daimi.au.dk/jwig-cnn/dSem/>.