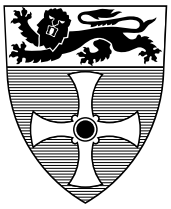
UNIVERSITY OF
NEWCASTLE



University of Newcastle upon Tyne

COMPUTING SCIENCE

Comments on several years of teaching of modelling programming language concepts

J W Coleman, N P Jefferson, and C B Jones

TECHNICAL REPORT SERIES

No. CS-TR-978 July, 2006

TECHNICAL REPORT SERIES

No. CS-TR-978 July, 2006

Comments on several years of teaching of modelling programming language concepts

J W Coleman, N P Jefferson, and C B Jones

Abstract

This paper describes an undergraduate course taught at the University of Newcastle upon Tyne titled Understanding Programming Languages. The main thrust of the course is to understand how language concepts can be modelled and explored using semantics. Specifically, structural operational semantics (SOS) is taught as a convenient and light-weight way of recording and experimenting with features of procedural programming languages. We outline the content, discuss the contentious issue of tool support and relate experiences.

Printed and published by the University of Newcastle upon Tyne,
Computing Science, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.

Bibliographical details

COLEMAN, J. W., JEFFERSON, N. P., JONES, C. B..

Comments on several years of teaching of modelling programming language concepts
[By] J. W. Coleman, N. P. Jefferson, C. B. Jones.

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2006.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-978)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-978

Abstract

This paper describes an undergraduate course taught at the University of Newcastle upon Tyne titled Understanding Programming Languages. The main thrust of the course is to understand how language concepts can be modelled and explored using semantics. Specifically, structural operational semantics (SOS) is taught as a convenient and light-weight way of recording and experimenting with features of procedural programming languages. We outline the content, discuss the contentious issue of tool support and relate experiences.

About the author

Joey earned a BSc (2001) in Applied Computer Science at Ryerson University in Toronto, Ontario. With that in hand he stayed on as a systems analyst in Ryerson's network services group. Following that he took a position at a post-dot.com startup as a software engineer and systems administrator. Having decided that research was likely more interesting than what he had been doing, Joey moved to Newcastle and earned a MPhil (2005) in Computing Science, and is currently working part-time on his PhD. He is involved primarily with the EPSRC "Splitting (Software) Atoms Safely" project, working on atomicity in software development methods. He is also involved in the RODIN project, working on methodology. Other associations include the DIRC project. His main interests lie in language design and semantics.

Nigel is currently studying for a PhD at the CSR, attached to the DOTS project under the supervision of Dr. Steve Riddle and funded by an EPSRC studentship. His area of research targets the reuse of black-box software COTS (commercial off the shelf) components and focuses on the formal semantics of component based languages. Before undertaking his PhD, he joined CSR in June 2002 as a General Duties Assistant after completing his BSc (Hons) in Computer Science at the University of Newcastle upon Tyne.

Cliff Jones is currently Professor of Computing Science and Project of the IRC on “Dependability of Computer-Based Systems”. He has spent more of his career in industry than academia. Fifteen years in IBM saw among other things the creation with colleagues in Vienna of VDM. Cliff is a fellow of the BCS, IEE and ACM. He Received a (late) Doctorate under Tony Hoare in Oxford in 1981 and immediately moved to a chair at Manchester University where he built a strong Formal Methods group which among other projects was the academic partner in the largest Alvey Software Engineering project (IPSE 2.5 created the "mural" theorem proving assistant). During his time at Manchester, Cliff had an SRC 5-year Senior Fellowship and spent a sabbatical at Cambridge with the Newton Institute event on "Semantics". Much of his research at this time focused on formal (compositional) development methods for concurrent systems. In 1996 he moved to Harlequin directing some 50 developers on Information Management projects and finally became overall Technical Director before leaving to re-join academia in 1999. Cliff's interests in formal methods have now broadened to reflect wider issues of dependability.

Suggested keywords

STRUCTURAL OPERATIONAL SEMANTICS,
FORMAL METHODS,
EDUCATION

Comments on several years of teaching of modelling programming language concepts*

J W Coleman, N P Jefferson, and C B Jones

School of Computing Science
Newcastle University
NE1 7RU, UK
e-mail: {j.w.coleman,n.p.jefferson,cliff.jones}@ncl.ac.uk

Abstract. This paper describes an undergraduate course taught at the University of Newcastle upon Tyne titled *Understanding Programming Languages*. The main thrust of the course is to understand how language concepts can be modelled and explored using semantics. Specifically, structural operational semantics (SOS) is taught as a convenient and light-weight way of recording and experimenting with features of procedural programming languages. We outline the content, discuss the contentious issue of tool support and relate experiences.

1 Introduction

The course discussed in this paper[1] is entitled "Understanding Programming Languages".[2] (For brevity, the course is referred to below by its number "CSC334".) It teaches the modelling of *concepts* from programming languages. Formally, it covers operational semantics using parts of VDM for the formal notation (including an unconventional emphasis on "abstract syntax" (see Section 3)) but tries not to labour the formalism itself. The teaching objectives are about the student being able to read a formal (operational) semantics and to experiment with language ideas by sketching a model.

The School of Computing Science at University of Newcastle Upon Tyne offers several undergraduate "degree programmes" each of which includes CSC334 as an optional final year course. The course is taught to a wide variety of students with varying degrees of experience with formal methods.

There are no prerequisite courses for the CSC334 course, but students from most degree programmes take compulsory 2nd year courses that teach VDM as an introduction to formal methods (the textbook used for this is [2]). Interestingly, the School of Computing Science does not offer a course on compilers and this has to be taken into account in the delivery of CSC334.

2 Instructor's Motivation

There are several reasons for teaching formal semantics at undergraduate level. Probably the strongest can be motivated from the "half life of knowledge" that can be im-

* This paper is published in the proceedings of the *Formal Methods in the Teaching Lab* workshop that happened at the Formal Methods 2006 conference in Hamilton, Ontario. Please cite that version instead of this technical report.

[1] A longer version of this paper can be found in [1].

[2] A book with the same title is being written.

parted: programming languages come and go — over previous ten year periods, there have been complete changes in the fortunes of one or another language (e.g. Pascal, Modula-n, Ada, C, C++ and Java just within main line procedural languages). Any language that we teach in a university course today might be added to the list of faintly remembered languages in a decade's time. It therefore behoves academics to try to teach something which will last longer and give students a way to look at future languages. There are of course very good books on comparative languages (a recent example is [3]). The fact that so many of the programming languages –even those which are widely used– exhibit really bad design decisions is also worrying and indicates that there is a need to give future computer scientists ways to explore ideas more economically than by building a compiler.

The idea of teaching students a way to *model* concepts in programming languages is attractive in itself but it also provides an opportunity to say things about the fundamental nature of Informatics. Computing science is not a natural science in which one is stuck with modelling the universe as it exists; neither is it usefully viewed as a branch of mathematics as one cannot ignore what can be realized in an engineering sense. This tension is nowhere more clearly seen than in the design of programming languages. Language designers must find a compromise between clarity of expression of programs written *in* the language and reasonable performance of implementations *of* the language. Of course, this list could be extended to include all sorts of issues like the ability to diagnose programmers' errors but the essential tension is that indicated above.

To actually model these concepts we use operational semantics. The essence of operational semantics is that it provides what John McCarthy called an "abstract interpreter" for the language under study. Both words are important. An interpreter makes clear how programs are executed; for an imperative language, it shows how statements cause changes to the state of the computation. The importance of this being described "abstractly" cannot be overemphasized: the interpretation can be understood and reasoned about because it is presented in terms of abstract objects.

This interpretative framework allows the students to reason about the language in terms of the overall system. This includes the intermediate configurations generated during a system's operations, and is in sharp contrast to semantic definitions that are defined only in terms of a mapping from inputs to outputs.

The course then explores language definition questions. Concerns about the separation of syntactic and semantic issues, and the problems inherent in extending a language with new features are handled; practical coursework is used to illustrate how interactions between language features can have deep structural implications for the language. The nature of procedural languages is covered followed later in the course by object-oriented languages, and the two styles are contrasted by examining the roles procedures and objects play. Typing of values and variables in a language is handled mostly at the "static-checker" level, unfortunately the scope of the course does not permit a thorough coverage of the error-handling techniques required for dynamic typing. The topic has, however, come up during the practical sessions with some frequency. Lastly, the link between programs and data in the overall system has been covered as time permitted.[3]

[3] Though we do not go into the LISP-like notions of programs as data, some students have made the connection independently.

3 Technical material covered

Because the interest is in *modelling* rather than the meta-theory of semantics, the course teaches by example. A series of three language definitions are tackled: $Base$, $Blocks$, and $COOL$.

- $Base$ introduces the basic idea of states and abstract interpretation; after beginning with a simple deterministic language, concurrency is used to explain the need to cope with non-determinism; a trivial (and rather dangerous) form of threads with sequences of unguarded assignments is modeled using "Plotkin rules" (see Section 4)
- $Blocks$ includes Algol-like blocks and procedures; it is used to show how the key idea of an "environment" can be employed to model sharing and the normal range of parameter passing mechanisms are discussed
- $COOL$ is a concurrent object-based language; this is where the rule form of description really pays off. The language is rich enough to explore many alternatives.

The natural division of discussing syntax and semantics (and the difficult to place issue of context dependencies) is used. Before addressing the semantics of a language, it is necessary to delimit the language to be described. A traditional concrete syntax defines the strings of a language and suggests a parsing of any valid string. The publication of ALGOL-60 [4] solved the problem of documenting the syntax of programming languages: "(E)BNF" offers an adequate notation for defining the set of strings of a language. Most texts on semantics are content to write semantic rules in terms of concrete syntax. Although this is convenient for small definitions, it really does not scale up to larger languages. We therefore base everything on *Abstract Syntax* descriptions, and in particular, we use VDM-style records to define the structure of the language.

Using abstract syntax has the advantage of immediately getting the students to think about the information content of a program rather than bothering about the marks inserted just as parsing aids. There is an additional bonus that pattern matching with abstract objects gives a nice way of defining functions and rules by separating the definitions into cases.

The class of $Programs$ defined by any context free syntax (concrete or abstract) is too large in the sense that things like type constraints are not required to hold. There are many ways of describing *Context Conditions* but we prefer to write straightforward recursive predicates over abstract programs and static environments rather than, for example, use type theory as in [5].

So, given a class of "well formed" abstract programs, how do we give the semantics? McCarthy's formal description of "micro-ALGOL" [6] defines an "abstract interpreter" which takes a $Program$ and a starting state and delivers the final state. This "abstract interpreter" is defined in terms of recursive functions over statements and expressions.

We have taught both recursive functions and Plotkin rules in the course as means of defining the semantics of the language. However, as of this past year we have dropped them in favour of focusing solely on Plotkin rules.

4 Plotkin rules

Non-determinism arises in many ways in programming languages. Certainly the most interesting cause is concurrency but it is also possible to illustrate via non-deterministic

constructs like Dijkstra's "guarded commands". Unfortunately, McCarthy's idea to present an abstract interpreter by recursive functions does not easily cope with non-determinacy. Defining the recursive functions so that they produce a set of states is not convenient because of the bookkeeping requirements.

In 1981, Gordon Plotkin produced the technical report on "Structural Operational Semantics"[4] [9]. This widely photo-copied contribution revived interest in operational semantics.

The advantage of the move to such a rule presentation is the natural way of presenting non-determinacy. Many features of programming languages give rise to non-determinacy in the sense that more than one state can result from a given (program and) starting state. This natural expression extends well to concurrent languages. The advantage of the rule format appears to be that the non-determinacy has been factored out to a "meta-level" at which the choice of order of rule application has been separated from the link between text and states. For this reason, the complications of writing a function which directly defines the set of possible final states are avoided. Here is a case where the notation used to express the concept of relations (on states) is crucial.

5 Tool support

A key question for teaching CSC334 has been the use of tool support. Tool support has only been used in the teaching of CSC334 during some of the years it has been offered, and is actually an addition of the second author. The inclusion of tool support has both deepened the understanding of some students, as well as increased the confusion for others. There is the extra burden of learning to use the tool as well as the differences between the tool's ASCII syntax and the classroom syntax. Because of this, the tool has been an optional but fully supported part of the course, and the choice of use was entirely left to the student.

The tool used is the CSK VDMTools® [10, 11] which many of the CSC334 students have experience of from other courses. It provides an environment in which a VDM specification may be syntax- and type-checked and explicit functions may be executed via an interpreter. The students are provided with language specifications translated into ASCII VDM-SL, notably with the semantic rules translated into functions so that they can be executed in the Toolbox interpreter. This translation, in some cases, produces functions that are significantly different than the original semantic rules that are taught in class.

Beyond the syntactic differences between the original semantic rules and their encoding in the tool, there are often cases where the two versions of a semantic rule or function are wildly different. This is most evident when translating an implicit definition: the tool cannot directly encode implicit definitions, so an explicit equivalent must be created. Unfortunately, the process of doing this often results in a large, ugly and confusing specification. It is of no practical benefit for the students to study and comprehend these explicit definitions; it is important that they focus on the meaning of the semantics and not the implementation issues. Because of this the students are shielded from much of the underlying explicit implementation by separating it from the main language specification through the use of mechanisms made available by the tool.

[4] This material, together with a companion note on its origins [7], has finally been published in a journal [8].

It is our belief that for some students at least, the benefits of using the tool outweigh the negatives. Through use of the tool, the students can easily identify bugs in their VDM syntax; quickly spot type errors in their specifications; and execute test programs to test and improve their understanding.

The vast majority of mistakes made are errors in the semantic definitions and therein lies the major benefit of using the tool. The execution of test programs highlights such semantic slips and allows greater understanding by directly showing students the consequences of their design decisions.

6 Pedagogic experience

This course is evaluated positively by the students who take it. As an optional course, they obviously tend to self-select and about one quarter of the potential cohort choose to pursue it. The limited number (approximately 20–40 students) makes it possible to adopt a reactive learning environment experimenting with ideas from the students.

The practical work of CSC334 is based heavily on problem solving. Threading through the semester is a large project to make non-trivial extensions to the provided language definitions, and the lecturer tries to keep things timed so that he is introducing concepts just before they are needed. The format of the course's final exam stresses problem solving: it is an open-book exam, and is based on a language specification included from the lectures.

One of the course's final events has evolved over the past few years. As initially run, a few of the students were chosen to study one of the language specifications used in the lectures[5], and they would have the chance to grill the lecturer on the choices made in the design of that language. Their role included gathering comments and questions from their classmates, though the unchosen students also had the opportunity to ask questions through the session. This walkthrough of the language had the side-effect of debugging the language design; the design errors found by the were very instructive.

This exercise transformed first into the lecturer redeveloping a portion of one of the specifications during the lectures, then in the following year, a larger portion of the specification was redeveloped. These lectures had several aims: eliciting direct student participation in the writing of the specification[6]; showing how errors are made during specification and how to both discover and correct them; and to give a real demonstration of the kind of thinking that is needed to do this kind of development — teaching directly by example. While the lecturer did have the language specification to hand, its use was kept to a minimum: mainly to keep the names of the variables synchronized with their notes.

There is, of course, much related material that could usefully be taught on semantics. Textbooks such as [12] and [13] provide excellent introductions to the basic notions of semantics, but –to our taste– do so without a practical context. Their concern with meta-properties of the language would motivate our students less well than experiments with modelling a range of programming language issues.

A preliminary analysis of the feedback from the students suggests that they consider the practical portion of the course to be the most effective in gaining an understanding

[5] The same specification that would be used in that year's exam.

[6] Mainly by continually asking the class what else was needed for a given rule.

of the core course ideas. We would conjecture that this arises from the problem-oriented nature of the course: the students are warned at the start of the course that the exam is a problem-based, and to pass the exam they will have to apply the course material.

From both the feedback as well as from discussions with the small groups during practical sessions it appears that the students agree with the notion that the course content has a longer "half-life" than language-specific details. Part of this, we believe, is the realization that design decisions made in languages can be quite arbitrary when there are several ways to model a given feature.

Tool support for this course is explored in greater depth in [1]; analysis of the related effects was omitted from this version as the last run of the course did not use the tool.

Acknowledgments All of the authors acknowledge the support of the EPSRC *DIRC* project. The first and third authors also acknowledge support from the EU IST-6 programme project *RODIN*, and the ESPRC project *"Splitting (Software) Atoms Safely"*. In addition the second author is grateful to the EPSRC funded *Diversity with Off-The-Shelf* (DOTS) project for providing his studentship.

References

1. Coleman, J.W., Jefferson, N.P., Jones, C.B.: Black tie optional: Modelling programming language concepts. Technical Report Series CS-TR-844, School of Computing Science, University of Newcastle Upon Tyne (2004)
2. Fitzgerald, J., Larsen, P.G.: Modelling systems: practical tools and techniques in software development. Cambridge University Press (1998)
3. Watt, D.A.: Programming Language Design Concepts. John Wiley (2004)
4. Backus, J.W., Bauer, F.L., Green, J., Katz, C., McCarthy, J., Naur, P., Perlis, A.J., Rutishauser, H., Samelson, K., Vauquois, B., Wegstein, J.H., van Wijngaarden, A., Woodger, M.: Revised report on the algorithmic language Algol 60. Communications of the ACM **6**(1) (1963) 1–17
5. Pierce, B.C.: Types and Programming Languages. MIT Press (2002)
6. McCarthy, J.: A formal description of a subset of ALGOL. In: [14]. (1966) 1–12
7. Plotkin, G.D.: The origins of structural operational semantics. Journal of Logic and Algebraic Programming **60–61** (2004) 3–15
8. Plotkin, G.D.: A structural approach to operational semantics. Journal of Logic and Algebraic Programming **60–61** (2004) 17–139
9. Plotkin, G.D.: A structural approach to operational semantics. Technical report, Aarhus University (1981)
10. CSK: VDMTools®: VDM-SL Toolbox Manual, www.vdmbook.com/tools.php. (2006)
11. CSK: VDMTools®: The CSK VDM-SL Language, www.vdmbook.com/tools.php. (2006)
12. Nielson, H.R., Nielson, F.: Semantics with Applications: A Formal Introduction. Wiley (1992) Available at http://www.daimi.au.dk/∼bra8130/Wiley_book/wiley.html.
13. Winskel, G.: The Formal Semantics of Programming Languages. The MIT Press (1993) ISBN 0-262-23169-7.
14. Steel, T.B.: Formal Language Description Languages for Computer Programming. North-Holland (1966)