Reprinted from: A. D. Gordon and A. M. Pitts (Eds), *Higher Order Operationational Techniques in Semantics* (Cambridge University Press, to appear), pp 1–8

Introduction

The articles in this volume concern operational semantics of higher order programming languages, mathematical techniques for developing the properties of such operational semantics, and applications of those techniques. In this Introduction we set the articles in the wider context of research into programming languages and bring out some of the themes and techniques that recur throughout the book.

Operational Semantics

The various approaches to giving meanings to programming languages fall broadly into three categories: denotational, axiomatic, and operational. In a denotational semantics, the meaning of programs is defined abstractly using elements of some suitable mathematical structure. In an axiomatic semantics, meaning is defined indirectly via the axioms and rules of some logic of program properties. In an operational semantics, the meaning of programs is defined in terms of their behaviour, for example the steps of computation they can take during program execution. Whilst the first two approaches have much in common with semantic studies in mathematical logic and linguistics, the third is distinctive of computer science. Nevertheless, operational semantics is often regarded as the 'poor relation' of the other two approaches. The reason for this has to do with the fact that it tends to be quite concrete, with important general properties of a language's various features sometimes obscured by the low-level description of how program execution takes place. Thus at its worst, an operational specification may come perilously close to committing the sin of 'definition by implementation' (which is a sin because such a definition is not particularly useful). Such a criticism was certainly pertinent in the early days of the subject and was a stimulus to the development of the more mathematically principled methods of denotational and axiomatic semantics. To quote Plotkin (1981, page 21), the early attempts at semantic definitions based on abstract machines had "a tendency to pull the syntax to pieces or at any rate to wander around the syntax creating various complex symbolic structures which do not seem particularly forced by the demands of the language itself".

This situation changed radically with the development by Kahn, Milner, Plotkin, and others of a *structural* approach to operational semantics (SOS). SOS is now very widely used and in particular features in all the contributions in this volume. Because of the seminal work on the subject by Plotkin (1981), SOS is often taken to entail the use of (labelled) transition systems to describe steps of computation. However, Milner and Kahn's uses of 'relational', 'natural', or 'big-step' semantics (of which the Definition of Standard ML by Milner, Tofte, Harper, and MacQueen (1997) is the shining example) were similar in spirit to Plotkin's development. In each case, there are two key ideas. First, inspired by their use in mathematical logic for specifying formal systems, inductive definitions are used to define relations between programs or machine configurations. These relations comprise the operational description of the language. Second, the rule schemes used in these inductive definitions are syntax-directed. This often permits a simplification of the configurations of the machine—in the best cases they become just certain phrases of the language itself. It means also that SOS has a compositional aspect more usually associated with denotational semantics and which is essential for reasoning about program properties.

Higher Order Programming Languages

By a *higher order* programming language we mean one in which the means of manipulation—be it function, or procedure, or process, or object—can itself be manipulated. Most of the papers in this volume make use of function-oriented languages based on various kinds of λ-calculi, both typed and untyped. The articles by Lassen, Mitchell-Hoang-Howard, and Sands concern pure functional languages; whereas those by Jeffrey, Morrisett-Harper, Pitts-Stark, Smith, and Talcott concern the combination of functions with various 'effects'—such as the dynamic creation and mutation of state, and the spawning of concurrent communicating threads of computation. Object-oriented languages are studied in the articles by Gordon and Smith.

Higher order languages present difficult challenges for any style of semantics, be it axiomatic, denotational, or operational. For example, an axiomatic semantics of higher order procedures which is sound, expressive and useable has proved elusive (see Reynolds (1982) and O'Hearn and Tennent (1993)). Another well-known example is the difficulty of finding mathematical structures which yield denotational semantics that are both compositional and agree with the observable behaviour of programs. The search for such 'fully abstract' denotational models has been long and hard even in the apparently simple case of pure higher order functions (Plotkin 1977), let alone for the combination of functions with local state (Meyer and Sieber 1988), or with features involving non-deterministic interaction (where even mere compositionality is a challenge). Recent developments involving mathematical games (Abramsky 1997; Hyland 1997) promise to solve many of these problems. As with the classical Scott-Plotkin denotational models based upon order-theoretic structure, a considerable mathematical investment is required to understand and apply such models.

The challenge of higher order languages for operational semantics is somewhat different. In contrast to the axiomatic and denotational approaches, it is relatively easy to give inductively defined, structural operational semantics for all of the language features mentioned above. The reader will find such definitions throughout the articles in this book. What is not so easy is to develop the mathematical properties and applications of such operational definitions. The articles in this book discuss various techniques for achieving such a development. A related problem is the syntax-dependent nature of operational methods, which makes it hard to reuse the theory developed for one language in a related language. The articles by Smith and Talcott address this issue.

Aims, Methods and Techniques

Of course, a principal aim of an operational semantics of a language is to give a mathematically precise definition of how to execute programs—for example as a guide to implementors of the language. Beyond that, the following three types of application are well represented in what follows.

Proving properties of programs. Operational semantics is frequently used to define notions of semantic equivalence of programs and to develop the theory of such notions. (Examples of this occur in the articles by Gordon, Jeffrey, Lassen, Pitts-Stark, and Talcott.)

Correctness of interpreters and compilers. Interpretations of one language in another occur throughout computer science and one naturally wishes to verify the extent to which meaning is preserved or reflected under such translations. (See the articles by Jeffrey, Morrisett-Harper, and Smith.)

Questions of efficiency. The concrete nature of an operational semantics makes it potentially[1] better suited to studying intensional questions of efficiency, use of temporary storage space, etc. (See the articles by Morrisett-Harper and Sands.)

In pursuit of the above aims, the following operational methods and techniques are employed in the articles in this volume.

Context lemmas and extensionality principles. For many languages (particularly deterministic ones), operational semantics gives rise to a natural notion of semantic equivalence variously called *contextual*, *operational*, or *observational* equivalence. Roughly speaking, two program phrases are contextually equivalent if occurrences of the phrases in any executable program, that is, in any 'context', can be interchanged without affecting the results of executing the program. Although this notion is intuitively appealing, for higher order languages it can be

[1]Computational complexity for higher order languages is a rather under-developed area.

difficult to establish its general properties (other than the fact that it is a congruence, that is, respects the language constructs, which follows immediately from its definition). For example, for a language with higher order functions, we may hope for an extensionality result that explains contextual equivalence of functions in terms of contextual equivalence of their results when applied to an argument. Milner's Context Lemma for the simply typed language PCF (Milner 1977; Plotkin 1977) is an early example of such a result: it says that one can restrict attention to simple, applicative contexts when proving contextual equivalence of PCF functions. For functional languages with effects, it seems that the best such context lemma we can hope for restricts attention to a rather more involved collection of contexts—the ones Mason and Talcott (1992) call 'closed instantiations of uses' (ciu). Various versions of this 'ciu' theorem of Mason and Talcott are treated in the articles by Talcott and Pitts-Stark.

Bisimulation. For languages involving interaction (between a program and its environment, or between concurrent threads within a program) contextual equivalence may provide a rather too coarse notion of semantic equivalence, identifying programs with rather different intermediate interactions. For this reason Milner (1989) and Park (1981) introduced a family of notions of semantic equivalence known collectively as *bisimilarities* and which have been highly influential in the development of process calculi. Such equivalences are co-inductively defined, that is, are the greatest fixed points of suitable monotone operations. Roughly speaking two processes are bisimilar if they are related by some bisimulation—a binary relation between processes with the property that whenever two processes are related and one can do an action, the other can match that action in such a way that the resulting processes remain related. Somewhat surprisingly, this notion was usefully fed back into the more traditional setting of functional programming by the work of Abramsky (1990) who obtains an extensional characterisation of contextual equivalence for his 'lazy λ-calculus' in terms of an applicative bisimilarity. In view of the results of Pitts (1994), one might expect such characterisations of contextual equivalence in cases where the denotational semantics involves reflexive domains. Indeed, the results of Howe (1996) and Gordon (1994, 1995) show that this is correct for various kinds of pure functional programming language. However, for functions with effects and for objects the situation is not so straightforward. The article of Pitts-Stark shows that such extensional characterisations of contextual equivalence cannot hold for call-by-value higher order functions with local state. Gordon's article, extending previous work of Gordon and Rees (1996), considers applicative bisimilarity for some object-based languages of Abadi and Cardelli (1996), together with a notion of *experimental equivalence* also of co-inductive character; he shows that the latter coincides with contextual equivalence whereas the former does not.

Howe's method for proving congruence. In the previous two paragraphs we have mentioned characterisations of contextual equivalence in terms of other, more extensional equivalences—'ciu' equivalence, bisimilarity, and experimental equivalence. The proof of such characterisations hinges upon showing that these other equivalences are congruences, that is, respect the various constructs of the language in question. Such congruence proofs are non-trivial for higher order languages (mainly because their operational semantics involve non-trivial instances of substitution). Howe (1996) originated a method for proving congruence that has proved to be remarkably robust, inasmuch as it has been adapted to a wide range of higher order languages. Here, Lassen provides a rather slick version of Howe's method for pure functional languages as part of his relational reasoning about contexts. Gordon applies Howe's method to both the applicative bisimilarity and the experimental equivalence he develops for the Abadi-Cardelli object calculi. And Jeffrey makes use of Howe's technique for a language of higher order functions with concurrent communication (a fragment of Reppy's CML).

Relational proof techniques. The results mentioned in the previous paragraph give useful proof methods for contextual equivalence, via the construction of suitable bisimulations. In the world of process calculi, this technique has been sharpened by the development of various auxiliary notions, such as 'bisimulation up to bisimilarity', and 'bisimulation up to context' (see particularly Sangiorgi (1994)). Lassen's article shows that, with some effort, similar techniques can be established, and are useful, for higher order functional languages. He also touches upon *improvement theory*, introduced by Sands (1995) and developed in detail in Sands' article in this volume. One functional program is an improvement of another, in Sands' sense, if its execution is more efficient (takes less steps) in any program context. This notion not only permits some reasoning about the time behaviour of recursive functional programs, but as Sands shows, it also provides a useful method for proving properties of contextual equivalence of such programs.

Analogues of domain-theoretic methods. Some of the proof techniques in this volume are direct analogues of familiar domain-theoretic techniques. Consider, for example, the familiar Tarski characterisation of the least fixed point $\mathit{fix}(f)$ of a continuous function f on a domain as the countable limit of $f^n(\perp)$. This has an operational analogue expressing a finiteness property of evaluation of recursively defined terms: see (Mason, Smith, and Talcott 1996) and the articles by Lassen and Pitts-Stark. Similarly, the calculus of finite elements in recursively defined domains (and in particular the 'minimal invariant' property of such domains) has an operational analogue which is explored in Smith's article. Such results enable one to reuse some familiar denotational methods, such as Scott's induction principle (Scott 1993). For example, the operational analogue of Scott induction is used in proving the Fundamental Property of the logical relation introduced in the article of Pitts-Stark. Birkedal and Harper (1997) provide another good example of the application of such techniques.

Methods from the λ-calculus. A glance through recent text books such as those by Gunter (1992) and Mitchell (1996) shows the extent to which the mathematics of the λ-calculus (both the typed and untyped varieties) has influenced the development of programming language theory. The Mitchell-Hoang-Howard article provides a good example of this: labelling techniques originally developed for the untyped λ-calculus are adapted to prove completeness of the usual reduction strategy for the functional language PCF and to prove a confluence result for a language with subtyping. Another example is provided by the notion of 'logical relation', originally developed for the simply typed λ-calculus by Plotkin (1973). Subsequently, versions of this notion have proved very useful in denotational semantics (the work of Reynolds (1974) is a classic example) and, more recently, in operational semantics (see Pitts (1997), for example). The article by Pitts-Stark provides an example of the use of logical relations in operational semantics, in this case for reasoning about contextual equivalence of functions with local state in an ML-like language.

References

Abadi, M. and L. Cardelli (1996). *A Theory of Objects*. Springer-Verlag, New York.

Abramsky, S. (1990). The lazy λ-calculus. In D. A. Turner (Ed.), *Research Topics in Functional Programming*, Chapter 4, pp. 65–117. Addison Wesley.

Abramsky, S. (1997). Semantics of interaction: an introduction to game semantics. In A. M. Pitts and P. Dybjer (Eds.), *Semantics and Logics of Computation*, Publications of the Newton Institute, pp. 1–31. Cambridge University Press.

Birkedal, L. and R. Harper (1997). Relational interpretation of recursive types in an operational setting (Summary). In *Proc. TACS'97*, Lecture Notes in Computer Science. Springer-Verlag, Berlin. to appear.

Gordon, A. D. (1994). *Functional Programming and Input/Output*. Distinguished Dissertations in Computer Science. Cambridge University Press.

Gordon, A. D. (1995). Bisimilarity as a theory of functional programming. In *Eleventh Conference on the Mathematical Foundations of Programming Semantics, New Orleans, 1995*, Volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier.

Gordon, A. D. and G. D. Rees (1996, January). Bisimilarity for a first-order calculus of objects with subtyping. In *Conference Record of the 23rd ACM Symposium on Principles of Programming Languages, St Petersburg Beach, Florida*, pp. 386–395. ACM Press.

Gunter, C. A. (1992). *Semantics of Programming Languages: Structures and Techniques*. Foundations of Computing. MIT Press.

Howe, D. J. (1996, February). Proving congruence of bisimulation in functional programming languages. *Information and Computation 124*(2), 103–112.

Hyland, J. M. E. (1997). Game semantics. In A. M. Pitts and P. Dybjer (Eds.), *Semantics and Logics of Computation*, Publications of the Newton Institute, pp. 131–184. Cambridge University Press.

Mason, I. A., S. F. Smith, and C. L. Talcott (1996). From operational semantics to domain theory. *Information and Computation 128*(1), 26–47.

Mason, I. A. and C. L. Talcott (1992). References, local variables and operational reasoning. In *Proceedings of the 7th Annual Symposium on Logic in Computer Science*, pp. 186–197. IEEE Computer Society Press.

Meyer, A. and K. Sieber (1988). Towards fully abstract semantics for local variables. In *Proc. 15th Symp. on Principles of Programming Languages, San Diego*, pp. 191–203. ACM.

Milner, R. (1977). Fully abstract models of typed lambda-calculi. *Theoretical Computer Science 4*, 1–22.

Milner, R. (1989). *Communication and Concurrency*. Prentice Hall.

Milner, R., M. Tofte, R. Harper, and D. MacQueen (1997). *The Definition of Standard ML (Revised)*. MIT Press.

Mitchell, J. C. (1996). *Foundations for Programming Languages*. Foundations of Computing. MIT Press.

O'Hearn, P. W. and R. D. Tennent (1993). Semantical analysis of specification logic, 2. *Information and Computation 107*, 25–57.

Park, D. (1981). Concurrency and automata on infinite sequences. In P. Deussen (Ed.), *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, Volume 104 of *Lecture Notes in Computer Science*, pp. 167–183. Springer-Verlag, Berlin.

Pitts, A. M. (1994). A co-induction principle for recursively defined domains. *Theoretical Computer Science 124*, 195–219. (A preliminary version of this work appeared as Cambridge Univ. Computer Laboratory Tech. Rept. No. 252, April 1992.).

Pitts, A. M. (1997). Reasoning about local variables with operationally-based logical relations. In P. W. O'Hearn and R. D. Tennent (Eds.), *Algol-Like Languages*, Volume 2, Chapter 17, pp. 173–193. Birkhauser. First appeared in *Proceedings Eleventh Annual IEEE Symposium on Logic in Computer Science*, Brunswick, NJ, July 1996, pp 152–163.

Plotkin, G. D. (1973, October). Lambda-definability and logical relations. Memorandum SAI-RM-4, School of Artificial Intelligence, University of Edinburgh.

Plotkin, G. D. (1977). LCF considered as a programming language. *Theoretical Computer Science 5*, 223–255.

Plotkin, G. D. (1981). A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University.

Reynolds, J. C. (1974). On the relation between direct and continuation semantics. In J. Loeckx (Ed.), *Automata, Languages and Programming, Proceedings 1974*, Volume 14 of *Lecture Notes in Computer Science*, pp. 141–156. Springer-Verlag, Berlin.

Reynolds, J. C. (1982). Idealized Algol and its specification logic. In D. Néel (Ed.), *Tools and Notions for Program Construction*, pp. 121–161. Cambridge University Press.

Sands, D. (1995). A naïve time analysis and its theory of cost equivalence. *Journal of Logic and Computation 5*, 495–541.

Sangiorgi, D. (1994, August). On the bisimulation proof method. Technical Report ECS-LFCS-94-299, Department of Computer Science, Edinburgh University.

Scott, D. S. (1993). A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science 121*, 411–440.