to this richness seriously undermines any appeal that the book would have for practitioners.

It is not easy to describe the readers who will gain most from reading this book. For instance, it does not contain a range of radically new theorems about advanced topics in program semantics or a comprehensive reference work that covers known semantic techniques. To the authors' credit, the preface essentially says it was not intended to be either of these things. To briefly summarise, this book succeeds in offering an alternative presentation of logic and predicate based program semantics, from a distinctive, thought-provoking point of view. As such, it is perhaps of greatest relevance to those with a serious interest in presentations of program semantics and underlying theories.

Brian MONAHAN
Department of Computer Science
University of Manchester
Manchester, United Kingdom


This book is a revised version of Peter Lee's doctoral dissertation, completed under the supervision of Uwe Pleban at the University of Michigan. Lee's thesis is that realistic compilers can be generated from denotational semantic specifications, provided that the latter are well structured for the purpose.

"Traditional" (Oxford-style) denotational specifications are not well structured, as Lee convincingly demonstrates. They specify mappings from programs to $\lambda$-expressions, which must then be evaluated using $\beta$-reduction. This is very expensive, whether evaluation is done entirely at run-time or partial evaluation is done at compile-time, because of the large number of closures present. "Classical" semantic-directed compiler generators, such as Mosses' SIS, Paulson's PSP, and Wand's SPS, are therefore prohibitively inefficient. Lee argues that this inefficiency is inherent. He also picks up and amplifies Mosses' argument that traditional denotational specifications are badly structured in another sense: they are difficult to read, to write, and to modify. Small extensions to the specified language often entail a change in the semantic model, forcing the entire specification to be rewritten. For example, adding loop exits would force a direct-type semantics to be rewritten in the continuational style.

Lee's proposal is to separate a language's semantic specification into two levels, the macro-semantics and micro-semantics. The interface between these levels is a semantic algebra, whose sorts might include:
IACTION (imperative actions, updating the store)
VACTION (value-yielding actions),

and whose operations might include:

NullStmt : IACTION
  (action that does nothing),
Assign : VACTION × VACTION → IACTION
  (assignment action),
StmtSeq : IACTION × IACTION → IACTION
  (statement sequencing action),
While : VACTION × IACTION → IACTION
  (while-statement action).

A language’s macro-semantics specifies a mapping from
example:

\[ \mathcal{S} : \text{stmt} \rightarrow \text{ENV} \rightarrow \text{IACTION}, \]
\[ \mathcal{S}[S_1, S_2]_{\text{env}} = \text{StmtSeq}(\mathcal{S}[S_1]_{\text{env}}, \mathcal{S}[S_2]_{\text{env}}), \]

where \text{env} is the static environment.

The micro-semantics in turn specifies the meaning of the actions. Several complementary micro-semantic specifications can be given, typically a continuational specification, an interpretive specification, and a code-generator specification. The clean separation of the macro- and micro-semantics insulates the semantic equations from changes in the semantic model.

Lee’s compiler generator, MESS, consists of the following components:

- The front-end generator turns a syntactic specification into a conventional syntactic analyzer. This module will translate each source program into an abstract syntax tree.
- The macro-semantics analyzer turns a macro-semantic specification into a “compiler core”. This module will perform contextual analysis, and translate the abstract syntax tree into a “prefix-form operator term”, a tree whose internal nodes correspond to the operations of the semantic algebra.
- The micro-semantics analyzer turns a micro-semantic specification into a code generator or interpreter.

When the micro-semantic specification is a code-generator specification, the generated compiler has quite a conventional structure, with the prefix-form operator term acting as an intermediate representation for the code generator.

Lee has used MESS to generate compilers for two imperative languages, one of which is quite large. He gives figures demonstrating that these compilers compare well with commercial (nonoptimizing) compilers in terms of object code quality. He is more reticent about compilation speed: the generated compilers seem to be rather slow (but still much faster than the products of classical semantics-directed compiler generators).
Lee uses the term *high-level semantics* to describe his method of specifying a language's semantics. I think it would be more accurate to characterize it as a well-engineered compiler description language. It is not really a satisfactory way of specifying a language's semantics, for the following reasons. First, a language's macro-semantic specification tends to be cluttered with details of the static semantics. Secondly, it does not truly specify the dynamic semantics, but only a mapping to the semantic algebra. The latter is not fixed, but may vary (perhaps subtly) from one language's specification to another. So the reader must consult a micro-semantic specification, which (Lee's own examples suggest) is likely to be hard to understand.

The idea of stratifying semantic specifications using semantic algebras was originally conceived by Peter Mosses. It is interesting to compare Lee's development of this idea with the parallel development of *action semantics* by Mosses and Watt. The latter recognizes the importance of using a *standard* semantic algebra, which can be designed and specified (and learned) once and for all, then used to specify many languages' semantics. This makes a more satisfactory basis for specifying semantics, and perhaps in the long term for compiler generation too. But of course Lee's compiler generator is available now, and is testimony to a truly remarkable postgraduate student project.

David A. Watt

*Computing Science Department*

*The University*

*Glasgow, United Kingdom*


This book is a collection of five papers written by participants in the ESPRIT 415 "Parallel Architectures and Languages for Advanced Information Processing—a VLSI-directed Approach" project. Its aim was to present a representative selection of the work performed within the 415 project by the Working Group on Semantics. A companion volume—*Parallel Computers: Object-Oriented, Functional and Logic*, edited by P. C. Treleaven—gives an overview of work on parallel architectures carried out in the same project.

Providing a detailed review of this wide-ranging book requires in-depth knowledge of parallel architectures, functional, logic and object-oriented programming languages and their implementation, semantics, specification and proof techniques. In the absence of this, I will instead summarise each chapter and then give some overall comments.

Chapter 1 (by P. H. M. America and J. J. M. M. Rutten) describes the Pool2 parallel object-oriented language. It introduces the language, gives an example