Hans Hüttel

Transitions and Trees

An Introduction to Structural Operational Semantics

CAMBRIDGE

CAMBRIDGE

more information - www.cambridge.org/9780521197465

Transitions and Trees

An Introduction to Structural Operational Semantics

Structural operational semantics is a simple, yet powerful mathematical theory for describing the behaviour of programs in an implementation-independent manner. This book provides a self-contained introduction to structural operational semantics, featuring semantic definitions using big-step and small-step semantics of many standard programming language constructs, including control structures, structured declarations and objects, parameter mechanisms and procedural abstraction, concurrency, non-determinism and the features of functional programming languages. Along the way, the text introduces and applies the relevant proof techniques, including forms of induction and notions of semantic equivalence (including bisimilarity).

Thoroughly class-tested, this book has evolved from lecture notes used by the author over a 10-year period at Aalborg University to teach undergraduate and graduate students. The result is a thorough introduction that makes the subject clear to students and computing professionals without sacrificing its rigour. No experience with any specific programming language is required.

HANS HÜTTEL is Associate Professor in the Department of Computer Science at Aalborg University, Denmark.

Transitions and Trees

An Introduction to Structural Operational Semantics

HANS HÜTTEL
Aalborg University, Denmark

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore,
São Paulo, Delhi, Dubai, Tokyo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521197465

First published 2010

Printed in the United Kingdom at the University Press, Cambridge

A catalogue record for this publication is available from the British Library

ISBN 978-0-521-19746-5 Hardback
ISBN 978-0-521-14709-5 Paperback

Additional resources for this publication at www.operationalsemantics.net

Contents

Preface

About this book

This is a book about structural operational semantics; more precisely it is a book that describes how this approach to semantics can be used to describe common programming language constructs and to reason about the behaviour of programs.

The text grew out of the lecture notes that I have used over a period of more than 10 years in the course *Syntax and semantics* which is taught to all students following the various degree programmes in computer science at Aalborg University. What began as a 10-page set of notes in Danish is now a textbook in English.

The book also includes chapters on related material, namely short introductions to type systems, denotational semantics and the mathematics necessary to understand recursive definitions.

Related work

This work was inspired by lecture notes by Plotkin (1981) (also written in Denmark), where this approach to programming language semantics was first presented.

The topic of structural operational semantics also appears in later books, three of which I will mention here.

Reynolds' book (Reynolds, 1999) is an excellent text that covers some of the same topics as this book but uses denotational and axiomatic semantics as well as structural operational semantics.

The book by Winskel (1993) is another very good textbook that covers many of the same topics as Reynolds' book.

Finally, I should mention Nielson and Nielson (2007), which introduces

and relates denotational, axiomatic and structural operational semantics and then gives an introduction to how these can be used in connection with static program analysis.

The book that you are now reading differs from the ones just mentioned in three important ways. First, the main topic is exclusively that of structural operational semantics. Second, both Reynolds and Winskel introduce domain theory early on; this book aims at developing the theory of structural operational semantics and making use of it with mathematical prerequisites of a more modest nature. Third, unlike the book by the Nielsons, the focus here is not that of program analysis. Instead, it is on how operational semantics can be used to describe common features of programming languages.

What you need to know in advance

This text is *not* intended as an introduction to programming; if you are a reader expecting this to be the case, you will probably be disappointed! The ideal reader should already have some experience with programming in a high-level imperative programming language such as C, Java or Pascal.

Programming language semantics is a mathematical theory. Therefore, the reader should also have some mathematical maturity. In particular, you should be familiar with basic notions of discrete mathematics – sets, functions and graphs and the proof techniques of proof by induction and proof by contradiction. Chapter 2 gives a short overview of some of this material. There are several good textbooks that you can consult as a supplement; one that I would recommend in particular is Velleman's book (Velleman, 2006).

Ways through the book

The book falls into four parts. The first two parts must be covered in any course for which this book is the main text, since the contents of these first four chapters are necessary to understand the material in the rest of the book. After that, there are the following dependences:

- Chapter 7 and Chapter 9 are independent of each other but both extend the language introduced in Chapter 6,
- Chapter 8 assumes knowledge of the parallel operator introduced in Chapter 5,
- Chapter 10 on small-step semantics for procedures and blocks also assumes knowledge of Chapter 6,
- Chapter 11 assumes knowledge of Chapters 8 and 10 and finally

- Chapter 15 assumes knowledge of the contents of Chapter 14.

Problems and thoughts

To learn a mathematical subject, one should of course read the text carefully but also learn to apply the content. For this reason, there are quite a few problems scattered throughout the text. As a rule of thumb, a problem will appear at the point in the text where it becomes relevant. I have chosen this approach since I would like you, the reader, to focus on the connection between the problem and the context in which it appears. You can read most of the text without solving the problems but I encourage you to solve as many of them as possible. In some places, I have put problems that are important for understanding the text and they are then marked as such.

I have also introduced mini-problems which I call **A moment's thought**. Here, the idea is to make you think carefully about what you have just read. Do *not* read the text without finding the answers to these mini-problems.

Related resources

The book has its own website, `http://www.operationalsemantics.net`, which has more information, including hints to the mini-problems. The website also holds information about the Danish-language version of the book, *Pilen ved træets rod*, including how to obtain a copy of it.

Acknowledgements

This book grew out of the years I have spent teaching, so, first, I would like to thank the students who have lived with the various incarnations of this text over the years and have made many useful comments that have helped improve and shape its content and form.

Second, I want to thank the people who have inspired me to reflect on the task of teaching mathematical subjects and teaching in general over the years: Jens Friis Jørgensen, Steffen Lauritzen, Finn Verner Jensen, Anette Kolmos, Helle Alrø and Ole Skovsmose.

Third, I would like thank all those who helped me make this book a reality. My thanks go to the people and organizations who have kindly allowed me to use the pictures in Chapter 1 and to David Tranah from Cambridge University Press for his encouragement.

A number of colleagues read parts of the manuscript and provided me with lots of important feedback. Special thanks are due to Denis Bertelsen,

Morten Dahl, Ulrich Fahrenberg, Morten Kühnrich, Michael Pedersen, Willard Rafnsson and last, but by no means least, Gordon Plotkin.

On an entirely personal level, there are others who also deserve thanks. Over the years, I have come to know many inspiring people through my extracurricular activities in human rights activism and music and, most recently, through the extended family of sisters and brothers that I now have. I am very grateful for knowing you all.

Finally, and most importantly, I want to thank my wife Maria and our daughter Nadia for being in my life. This book is dedicated to you.

About the illustrations

- The picture of Alfred Tarski on p. 4 is by George M. Bergman and used courtesy of Wikimedia Commons under the GNU Free Documentation License.
- The picture of Dana Scott on p. 7 is used courtesy of Dana Scott.
- The picture of Christopher Strachey on p. 7 is used courtesy of Martin Campbell-Kelly.
- The picture of Gordon Plotkin on p. 8 is used courtesy of Gordon Plotkin.
- The picture of Robin Milner on p. 9 is used courtesy of Robin Milner.
- The picture of Tony Hoare on p. 9 is used courtesy of Tony Hoare.
- The picture of Joseph Goguen on p. 10 is used courtesy of Healfdene Goguen.
- The pictures of Ariane 5 on p. 12 are used courtesy of the ESA/CNES and are ©AFP/Patrick Hertzog, 1996.
- The picture of the Mars Climate Orbiter on p. 14 is used courtesy of NASA.

Illustrations

Tables

PART I

BACKGROUND

1

A question of semantics

The goal of this chapter is to give the reader a glimpse of the applications and problem areas that have motivated and to this day continue to inspire research in the important area of computer science known as programming language semantics.

1.1 Semantics is the study of meaning

Programming language semantics is the study of *mathematical models of and methods for describing and reasoning about the behaviour of programs.*

The word *semantics* has Greek roots[1] and was first used in linguistics. Here, one distinguishes among *syntax*, the study of the structure of languages, *semantics*, the study of meaning, and *pragmatics*, the study of the use of language.

In computer science we make a similar distinction between syntax and semantics. The languages that we are interested in are *programming languages* in a very general sense. The 'meaning' of a program is its behaviour, and for this reason programming language semantics is the part of programming language theory devoted to the study of *program behaviour.*

Programming language semantics is concerned only with purely internal aspects of program behaviour, namely what happens within a running program. Program semantics does not claim to be able to address other aspects of program behaviour – e.g. whether or not a program is user-friendly or useful.

In this book, when we speak of semantics, we think of *formal semantics*,

[1] The Greek word (transliterated) is *semantikós*, meaning 'significant'. The English word 'semantics' is a singular form, as are 'physics', 'mathematics' and other words that have similar Greek roots.

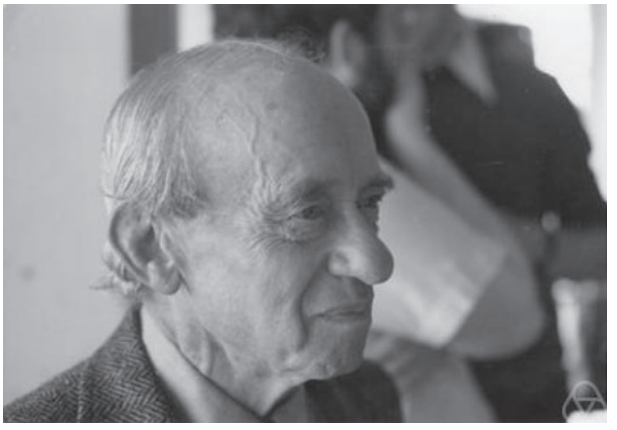

Figure 1.1 Alfred Tarski

understood as an approach to semantics that relies on precise mathematical definitions.

Formal semantics arose in the early twentieth century in the context of mathematical logic. An early goal of mathematical logic was to provide a precise mathematical description of the language of mathematics, including the notion of truth. An important contributor in this area was the logician Alfred Tarski (Figure 1.1) (Tarski, 1935).

Many of the first insights and a lot of the fundamental terminology used in programming language semantics can be traced back to the work of Tarski. For instance, the important notion of *compositionality* – that the meaning of a composite language term should be defined using the meanings of its immediate constituents – is due to him. So is the insight that we need to use another language, a *metalanguage*, to define the semantics of our target language.

1.2 Examples from the history of programming languages

The area of programming language semantics came into existence in the late 1960s. It was born of the many problems that programming language designers and implementors encountered when trying to describe various constructs in both new and existing programming language.

The general conclusion that emerged was that an informal semantics, however precise it may seem, is not sufficient when it comes to defining the behaviour of programs.

1.2.1 ALGOL 60

The programming language ALGOL 60 was first documented in a paper from 1960 (Backus and Naur, 1960), now often referred to simply as 'the ALGOL 60 report'. ALGOL 60 was in many ways a landmark in the evolution of programming languages.

Firstly, the language was the result of very careful work by a committee of prominent researchers, including John Backus, who was the creator of FORTRAN, John McCarthy, the creator of Lisp, and Peter Naur, who became the first Danish professor of computer science. Later in their careers, Backus, McCarthy and Naur all received the ACM Turing Award for their work on programming languages.

Secondly, ALGOL 60 inspired a great many subsequent languages, among them Pascal and Modula.

Thirdly, ALGOL 60 was the first programming language whose syntax was defined formally. The notation used was a variant of context-free grammars, later known as Backus–Naur Normal Form (BNF).

However, as far as the semantics of ALGOL 60 is concerned, Backus and his colleages had to rely on very detailed descriptions in English, since there were as yet no general mathematical theories of program behaviour. It turned out to be the case that even a group of outstanding researchers (who for the most part were mathematicians) could not avoid being imprecise, when they did not have access to a formalized mathematical theory of program behaviour. In 1963 the ALGOL 60 committee therefore released a revised version of the ALGOL 60 report (Backus and Naur, 1963) in which they tried to resolve the ambiguities and correct the mistakes that had been found since the publication of the original ALGOL 60 report.

However, this was by no means the end of the story. In 1967, Donald E. Knuth published a paper (Knuth, 1967) in which he pointed out a number of problems that still existed in ALGOL 60.

One such problem had to do with global variables in procedures. Figure 1.2 illustrates the nature of the problem. The procedure `awkward` is a procedure returning an integer value.

```
integer procedure awkward
begin comment x is a global variable
 x := x+1
 awkward := 3
end  awkward
```

Figure 1.2 An ALGOL 60 procedure. What is its intended behaviour?

The procedure `awkward` manipulates the value of a global variable and therefore has a side effect. However, the ALGOL 60 report does not explain whether or not side effects are allowed in procedures. Because of this, there is also no explanation of how arithmetic expressions should be evaluated if they contain side effects.

Let us consider a global variable `x` whose value is 5 and assume that we now want to find the value of the expression `x+awkward`. Should we evaluate `x` before or after we evaluate `awkward`? If we evaluate `x` first, the value of the expression will be 8; should we evaluate `x` after having called `awkward`, we get the value 9!

One consequence of Knuth's paper was that the ALGOL 60 committee went back to the drawing board to remove the ambiguities. The main reason why it took so long to discover these problems was that the language designers had no precise, mathematical criterion for checking whether or not all aspects of the language had been defined.

1.2.2 Pascal

Pascal, a descendant of the Algol family, was created by Niklaus Wirth and first documented in a book written with Kathleen Jensen (Jensen and Wirth, 1975). Ever since then, Pascal has been a common introductory language in computer science degree programmes around the world.

Even though great care was taken in the exposition of the language features, Pascal also suffers from the problems associated with an informal semantics. In particular, there are problems with explaining scoping rules – in fact, the scoping rules are barely explained in the book. There is mention of global variables; however, nowhere in the text is it explained what a global variable is, let alone what its scope should be. Nor are there any rules that specify that a variable must be declared before it is used!

All existing implementations of Pascal assume this (except for pointer variables), but the declaration-before-use convention is not part of the original definition of the language.

1.3 Different approaches to program semantics

The development of a mathematical theory of program semantics has been motivated by examples such as the ones given above. There are several ways of providing such a mathematical theory, and they turn out to be related.

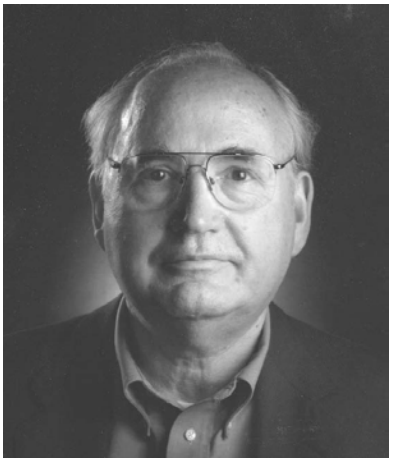
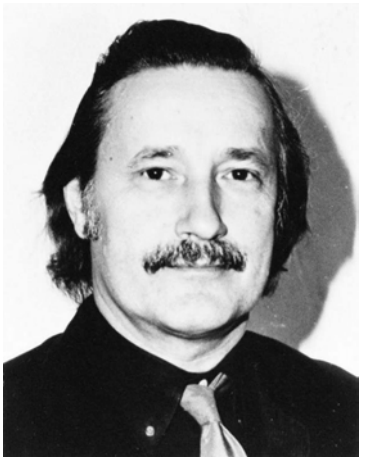

Figure 1.3 Dana Scott (left) and Christopher Strachey (right)

Denotational semantics was the first mathematical account of program behaviour; it arose in the late 1960s (Strachey, 1966, 1967; Scott and Strachey, 1971) and was pioneered by Dana Scott and Christopher Strachey (Figure 1.3), who at the time were both working at Oxford University.

In denotational semantics, the behaviour of a program is described by defining a function that assigns meaning to every construct in the language. The meaning of a language construct is called its *denotation*. Typically, for an imperative program, the denotation will be a *state transformation*, which is a function that describes how the final values of the variables in a program are found from their initial values.

Structural operational semantics – the main topic of this book – came into existence around 1980 and is due to Gordon Plotkin (Figure 1.4), who gave the first account of his ideas in a set of lecture notes written during his sabbatical at Århus University in 1980 (Plotkin, 1981). An important early contribution is that of Robin Milner (Figure 1.5), who used Plotkin's approach to give a labelled semantics to the process calculus CCS (Calculus of Communication Systems) (Milner, 1980). Plotkin (2004) gives a detailed account of the origins and early history of the area.

In structural operational semantics one specifies the behaviour of a program by defining a transition system whose transition relation describes the evaluation steps of a program. One of the underlying motivations for this approach was that it is possible to give a simple account of concurrent programs; previous attempts to give a semantic description of even simple

Figure 1.4 Gordon Plotkin

parallel programming languages had used denotational semantics and had turned out to be quite complicated.

A central insight of this approach, and one to which we shall return repeatedly throughout this book, is that one can describe the evaluation steps of a syntactic entity (such as a program) in a structural fashion, that is, by means of an inductive definition based on the abstract syntax.

Axiomatic semantics is due to Tony Hoare (Hoare, 1969; Apt, 1981) (Figure 1.6) and, like denotational semantics, it is a product of the late 1960s. Here one describes a language construct by means of mathematical logic. More precisely, one defines a set of rules that describe the assertions that must hold before and after the language construct has been executed.

Algebraic semantics is related to denotational semantics and describes the behaviour of a program using universal algebra (Guessarian, 1981; Goguen and Malcolm, 1996). The members of the research collective behind the OBJ specification language, with Joseph Goguen (Figure 1.7) as a prominent contributor, have been important figures in the development of this approach.

These four approaches to programming language semantics are not rivals. Rather, they complement each other. Some approaches are more suitable than others in certain situations. For instance, it is much easier to describe

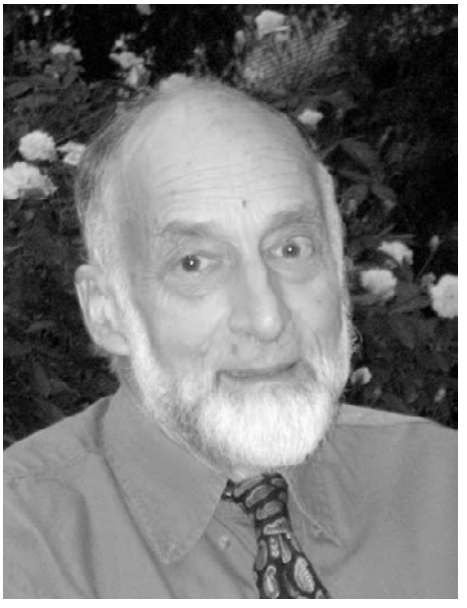

Figure 1.5 Robin Milner

Figure 1.6 Tony Hoare

parallel and nondeterministic program behaviour using structural operational semantics than by means of denotational semantics.

There are many precise mathematical results relating the four approaches. In this book we give an example of such a result in Chapter 15, where we

Figure 1.7 Joseph Goguen

show that the structural operational semantics and the denotational semantics of the **Bims** language are equivalent in a very precise sense.

1.4 Applications of program semantics

The area of program semantics has turned out to be extremely useful in situations where it is important to give a precise description of the behaviour of a program. Here are some prominent examples.

1.4.1 Standards for implementation

The formal semantics of a programming language is not meant as an alternative to the informal descriptions of programming constructs found in introductory programming textbooks. A formal semantics serves a very different purpose, namely to act as a yardstick that any implementation must conform to.

The examples mentioned in Section 1.2 all helped make computer scientists aware of the fact that *only a precise semantic definition can provide an exhaustive and implementation-independent account of all aspects of a programming language*. Such an account is particularly necessary if one is a 'superuser' of the language whose task is to implement an interpreter or a compiler or, in general, to create a language-dependent programming