Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

288

Andrzej Blikle

MetaSoft Primer

Towards a Metalanguage for Applied Denotational Semantics

Springer-Verlag

Berlin Heidelberg New York London Paris Tokyo

Author

Andrzej Blikle
Institute of Computer Science
Polish Academy of Sciences
PKiN P.O. Box 22, 00-901 Warsaw, Poland

CR Subject Classification (1987): D.3.1, D.2.10, F.3.2–3

ISBN 3-540-18657-3 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-18657-3 Springer-Verlag New York Berlin Heidelberg

Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
2145/3140-543210

FOREWORD

This book is devoted to a simplified version of denotational semantics, also known as *naive denotational semantics* (NDS), where sets are used in the place of Scott's reflexive domains and where jumps are described without continuations. The first public announcement of this approach dates back to [Blikle 82] and in a more complete version to [Blikle, Tarlecki 83]. Since then several experiments were undertaken in order to estimate the adequacy of NDS for applications. One group of experiments concerned the construction of models for typical software mechanisms such as procedures with parameters, blocks with local objects, jumps, escapes, exception handlers, pointers, user-defined types, error-handling mechanisms, concurrency. Another group was devoted to the construction of models for some typical software systems such as Pascal-like languages, Lisp-like languages, OCCAM^{TM}-like languages, word processors, operating systems or data-base management systems.

The mentioned experiments have proved that naive denotational semantics can be conveniently and rigorously used in applications. They also have led to the establishment of a kernel of a definitional metalanguage and of a general methodology of using denotational techniques in software design.

Our book is devoted to the former subject and consists of two parts. Part One starts from a general theory of chain-complete partially ordered sets (cpo's), continuous functions and their least fixed points. This theory provides a general mathematical framework for the recursive definitions of denotations, of their domains, of syntax and of the functions of semantics. Next we describe several calculi based on the cpo's of relations, functions, formal languages and domains, and we introduce a corresponding notation. Since the denotational definitions of software should serve not only the purpose of software specification, but also as a ground where one can prove the properties of software, we devote two sections to the introduction of appropriate logical tools. We introduce, and motivate, a calculus of three-valued predicates and then we discuss the derivation of Hoare's logic proof rules on that ground.

In Part two we show how to use the introduced mathematical tools in constructing a denotational definition of an existing programming language. We also show how to formulate and prove typical properties of the defined language and how to develop a corresponding program-correctness logic.

As a programming language in our example we have chosen Pascal. Since the size of the book does not permit to give a full definition of that language, we have restricted ourselves to its subset which contains basic commands and expressions and a (nearly) complete mechanism of arrays, records and pointers. On that ground we discuss the issue of user-definable types in programming languages and we show how to model them in NDS. We also point out how inadequate are the informal definitions of these mechanisms in the report of Pascal [Jensen, Wirth 78] and in the ISO standard of that language.

For the already mentioned sake of brevity we have excluded from our example both procedures and jumps – two software mechanisms which have stimulated the development of the so called *standard denotational semantics* (see Introduction). Readers interested in the description of these mechanisms within NDS, i.e. without the use of reflexive domains and continuations, should refer to [Blikle, Tarlecki 83]. We also omit methodological considerations which can be found in [Blikle 87]. It is argued there that in the process of software design one should develop denotations in the first place and then should derive syntax from them.

In the present form the notation introduced in Part One of our book can only be used in "handwritten" applications, i.e. without any specialized computer support. Although such applications also make sense – e.g. a formal definition of ADA (cf.[Bjørner,Oest 80]), which has later served in the development of ADA compiler in Dansk Datamatik Center, has been developed in that way – they require a sufficiently enthusiastic and well-trained team. Any broader use of denotational techniques in an industrial environment must be preceded by the development of a computer-support system consisting of a specialized editor, type checker, data-base facilities, rapid prototyping facilities and the like. This in turn requires the construction of a fully formalized definitional metalanguage with a sufficiently strong typing system, modularization techniques, proof support, etc. For the realization of that goal a five-year project **MetaSoft** has been initiated in Fall 1985 in the Institute of Computer Science of the Polish Academy of Science in Warsaw and Gdańsk. Part One of our book contains, therefore, the description of the semantic kernel of the future metalanguage of MetaSoft.

Our book is addressed to readers interested in the applications of denotational semantics: researchers who are developing projects similar to **MetaSoft**, software engineers interested in the formal methods of software specification, students of computer science departments. We hope that the book may also be used as a supplementary reading for university courses on applied denotational semantics and VDM. For that purpose a list of exercises has been included in Part One. The only prerequisites for the readers are the elementary set theory and logic, the elements of formal language theory and the ability of reading Pascal programs. A familiarity with standard denotational semantics, or with VDM, may help in appreciating our motivations but is not necessary.

At the end a few remarks about how to read this book. Since Part One introduces many mathematical constructions, not all of which are used in Part Two, readers interested primarily in applications may only glimpse through Part One in the first reading and return to it later. In that case in the first reading they may skip Sec.3, Sec.9 and Sec.10.

ACKNOWLEDGMENTS

My interest in applied denotational semantics started in Spring 1980 during a visit in Dines Bjørner's group at the Technical University of Denmark in Lyngby. Discussions about VDM and its applications inspired my research on a VDM-like denotational semantics. That work was continued in May/March 1981 at the University of Linkoeping in Sweden where I had the first opportunity of lecturing and discussing a continuation-free denotational semantics [Blikle 81b]. The ideas developed there and combined with a set-theoretic approach to domains were presented in June 1982 at an IFIP's W.G.2.2 group meeting in Garmisch-Partenkirchen. Andrzej Tarlecki contributed to them later which resulted in a common paper [Blikle, Tarlecki 83] presented as an invited lecture at the 1983 IFIP Congress in Paris.

The first draft of the present book was written in May 1983 during a visit in the Istituto di Scienze dell'Informazione of the University of Turin. Large parts of the material were discussed there in a series of seminars. Andrea Maggiolo-Schettini contributed to an (unpublished) exercise where a small database-management language was defined in the NDS framework. The manuscript prepared in Turin was revised and completed in August 1983 during my visit at the Department of Computer Science of the University of Manchester. Discussions with Cliff Jones and Derek Andrews substantially contributed to both the theoretical part of the book and to the example. This material was printed as [Blikle 83] in Pisa with the support of CNR Projetto P_1 Cnet and was later used as class notes for two of my courses: a course offered to a Special Interest Group for Formal Methods of Software Engineering of the Polish Computer Society in Warsaw in 1984, and another course given to students in the Institute of Datalogy of the University of Copenhagen in Winter

1985. The listeners of both courses have communicated to me many relevant remarks. I also especially appreciated the discussions which I had with Niel Jones in Copenhagen.

The present version of Part One of the book was completed partly during my visit at the University of Pisa in September 1985 and partly in March/May 1986 when I was visiting again Dines Bjørner's group in Lyngby. Sec.9 on three-valued predicates was also discussed in seminars in Dansk Datamatik Center with the members of project **RAISE**.

To all the institutions mentioned above I wish to express my gratitude for excellent conditions and atmosphere which they have created for my work. Special thanks are also addressed to the listeners of my seminars and courses for many stimulating discussions and remarks.

Although the major part of the book has been written when I was working outside Poland, large parts of the material were thoroughly discussed with my Polish colleagues. Here special thanks should be addressed to Stefan Sokołowski and Andrzej Tarlecki, today both in the **MetaSoft** group, who have read and discussed with me some early versions of this book. Also Marek Lao, Marek Ryćko and Ida Szafrańska have communicated interesting remarks. No need to say that the responsibility for all mistakes which remain in the book is entirely of the author.

The list of acknowledgments would have been incomplete if I did not mention excellent books [Gordon 79] and [Stoy 77] which have introduced me into the realm of denotations. Last but certainly not least, the inspiring influence of Dana Scott's famous works on fixed points, domains and lambda-calculus models cannot be overestimated.

Warsaw in August 1987 Andrzej Blikle

INTRODUCTION

The method of denotational semantics has been developed at the beginning of the decade 1970-80 as a mathematical technique of writing implementation- -independent definitions of software. Conceptually it has its roots in mathematical logic, where the meaning of an expression is a function and the meaning of a whole is a combination of the meanings of its parts. Technically, however, it has to deal with much more complicated mechanisms.

The pioneers of denotational semantics have felt particularly challenged by the problem of describing the mathematically most unnatural features of ALGOL-60: the self-applicability of procedures (a procedure may take itself as a parameter) and jumps nested in structured programs. The former problem has led to a model of **reflexive domains** [Scott 71] and [Scott 76], the latter to a technique of **continuations** [Strachey, Wadsworth 1974]. Their combination gave a powerful definitional method known today as **standard denotational semantics** (SDS). A full definition of ALGOL-60 in the SDS style was given in [Mosses 74].

Standard denotational semantics quickly became known and appreciated in the academic community. Its most important contribution to the formal specification of software consists in providing the first mathematical model for a compositional (i.e. inductively defined) semantics of complex programming languages and in stimulating broad research aimed at the applications of mathematics in software design.

Unfortunately the applications of (pure) SDS have remained rather limited. Although SDS formally provides an adequate ground where to define (old) and design (new) software, none of widely used programming languages has been designed in using SDS and only very few have been given SDS definitions later.

The discrepancy between the potential advantages of SDS and its actual applications has not been caused by the lack of interest of software engineers in formal methods. The main obstacles of broader applications of SDS have always been of a rather technical nature:

- the lack of a convenient notation (metalanguage) for real-life applications,
- the conceptual and technical complexity of reflexive domains and continuations.

The first major breakthrough in this situation was offered by Vienna Development Method (VDM) [Bjørner, Jones 78]. That method has provided a metalanguage, called META-IV, suitable for large denotational definitions and offering a technique of defining jumps without continuations [Jones 78]. The authors of VDM also decided to treat the Scott model of reflexive domains informally by assuming that for a practical purpose reflexive domains may be "regarded" as sets. At that point they were later supported by other authors who popularized denotational semantics, such as M.Gordon [Gordon 79] or J.Stoy [Stoy 80].

The philosophy and techniques of VDM proved to be useful in many applications. Formal definitions of PL/1, ALGOL-60, Standard Pascal, Pascal R, Pascal Plus, Modula-2, Edison, CHILL, Prolog and Ada have been written in the VDM style (see [Bjørner, Phren 83] for references). This resulted in a better understanding of these systems and in finding many ambiguities and inconsistencies in them. Some of these definitions, e.g. of Pascal Plus, Edison, CHILL and Ada, have been used later in the development of compilers. Several data-base systems (or their parts) were formally defined, e.g. the PL/1 programmers' interface to full concurrent System/R, System 2000 and CODASYL/DBTG. Some aspects of operating systems, office automation systems and the like have been defined in VDM style and were partly used later in implementations. All these examples have convinced many practitioners that denotational semantics may be a handy tool in real-life applications. Many universities started to offer regular courses on VDM. Some industrial institutions decided to consider VDM as their standard for software specification.

The vulgarization of Scott's model in VDM has very well paid in the applications where a formal definition of software is used only as a formalized reference-manual for human readers. It seems, however, questionable whether such a vulgarization can provide an adequate framework for proving the correctness of software or for the development of systems that support code generation. Besides, it is rather inconsistent to proclaim and advocate mathematical style and at the same time to agree for the violation of mathematical rules at the most critical point of the model. This may also lead to technical inconsistencies since reflexive domains and sets behave differently (form essentially different algebras), especially if fixed-point equations are concerned. Moreover, some domain constructors used in VDM, such as e.g. $A - B$, $A \overset{\sim}{\to} B$ or $A \underset{m}{\to} B$ are not definable for reflexive domains.

The theoreticians of VDM (cf.[Stoy 80]) suggest several technical remedies to these problems. In order to make the algebra of reflexive domains closer to the algebra of sets, some operations, like products, must be redesigned and some others, like subtraction, must be forbidden. One also has to forget about the difference between partial functions, total functions and mappings and represent all of them by continuous functions between cpo's.

All these technical restrictions are not very convenient in applications and therefore are not very rigidly observed by VDM users. For instance, one frequently relies on the fact that mappings are finite-domain functions and therefore a test whether an element belongs to the domain of a mapping is computable.

The discrepancy between the theory and practice of VDM puts a formal question-mark on the consistency of VDM definitions. On the other hand, when reading such definitions one usually has a strong impression that they are not inconsistent. In fact these definitions can most frequently be given consistent interpretations since in the majority of software systems one does not deal with self-applicability and therefore all semantic domains may be regarded as sets. This is obviously true for most operating systems, communication protocols, data-base management systems, spread-sheets, word processors, etc. This is also true for nearly all modern programming languages including Pascal, Modula, Ada, OCCAM and many others.

Self-applicability in programming languages appears essentially in only two standard situations:

1) if procedural recursion is elaborated dynamically, like in Lisp;

2) if a procedure may be directly or indirectly passed to itself as an actual parameter, like in Algol-60.

A glance on programming languages which were designed after 1970 shows a clear tendency to avoiding both these mechanisms. Static binding has been considered safer than dynamic binding and procedures are usually restricted in a way which protects them against self-applicability. For all such languages semantic domains may be just sets.

As was already mentioned in the Foreword, in this book we describe a kernel of a metalanguage for a set-theory based denotational semantics. We define and discuss several mathematical tools which are useful in constructing the denotational models of software in that style. The notation which we propose has been strongly influenced by META-IV. However, in contrast to the former, our metalanguage is going to be a pure functional language. The major extensions with respect to META-IV are binary relations, languages with infinite words, McCarthy's three-valued predicates and program-correctness statements.

CONTENTS