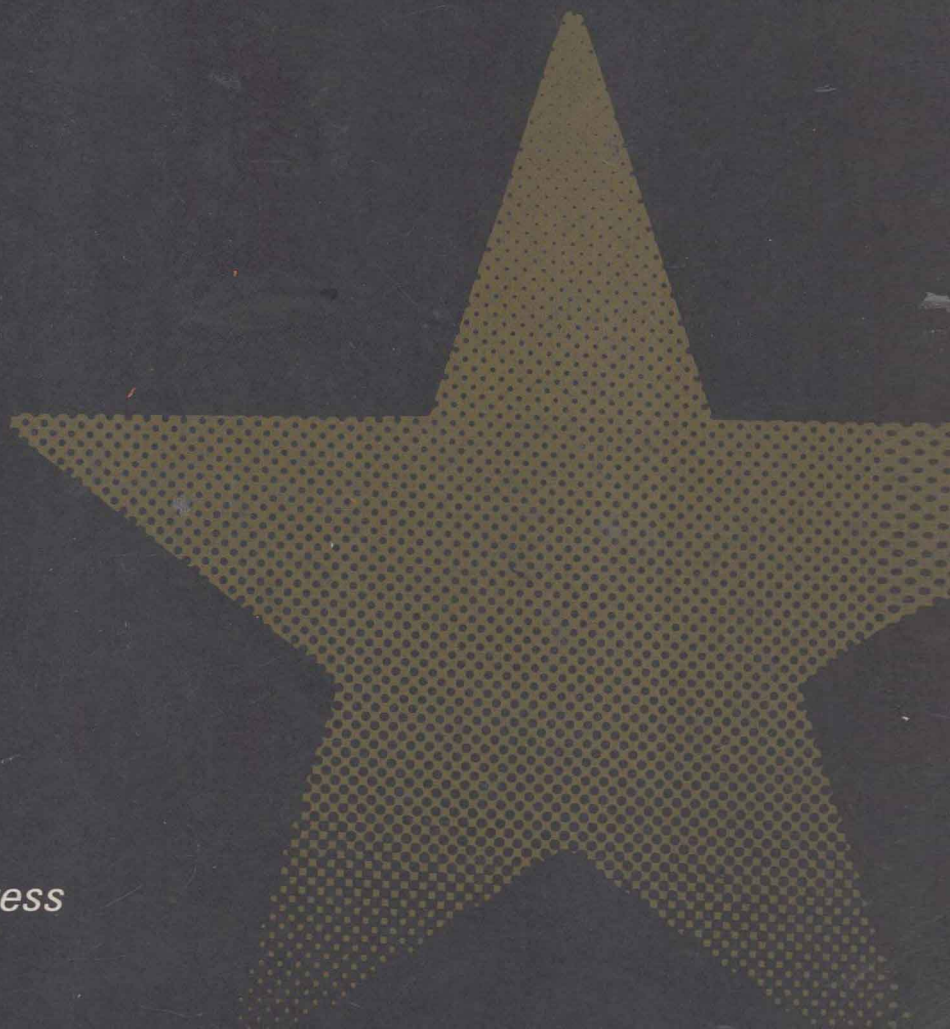
ACM Doctoral Dissertation Award 1986

Full Abstraction and Semantic Equivalence

Ketan Mulmuley



The MIT Press

Full Abstraction and Semantic Equivalence

Ketan Mulmuley

The MIT Press
Cambridge, Massachusetts
London, England

This dissertation was submitted in August 85 to the Department of Computer Science, Carnegie-Mellon University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy. The research reported in this document was supported in part by funds from the Computer Science Department of Carnegie-Mellon University, and by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, monitored by the Air Force Avionics Laboratory under Contract F33615-81-K-1539. The views and conclusions contained in it are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

The book was typeset by the author in LaTeX and printed on an Imagen Imprint-10 printer.

This book was printed and bound in the United States of America.

Library of Congress Cataloging-in-Publication Data

Mulmuley, Ketan.
Denotational and operational semantics.

(ACM doctoral dissertation awards; 1986)
Originally presented as the author's thesis (Ph. D.—Carnegie-Mellon University, 1985) under the title: Full abstraction and semantic equivalence.
1. Programming languages (Electronic computers)—Semantics. I. Title. II. Series: ACM doctoral disseration award; 1986.
QA76.7.M84 1987 005.13 86-33819
ISBN 0-262-13227-3

Full Abstraction and Semantic Equivalence

ACM Doctoral Dissertation Awards

1982

Area-Efficient VLSI Computation
by Charles Eric Leiserson

1983

Generating Language-Based Environments
by Thomas W. Reps

1984

Reduced Instruction Set Computer Architectures for VLSI
by Manolis G. H. Katevenis

1985

Bulldog: A compiler for VLIW Architectures
by John R. Ellis

1986

Full Abstraction and Semantic Equivalence
by Ketan Mulmuley

Computational Limitations of Small Depth Circuits
by Johan Håstad

To my parents

SERIES FOREWORD

The Doctoral Dissertation Award is presented each year by the Association for Computing Machinery (ACM). This award recognizes the best dissertation written in a computer-related field during the previous year. The winning author receives a cash award of $1000 from ACM. In addition, the winning dissertation is published by MIT Press, with the author receiving the appropriate royalties.

Since its inception in 1982, the award has gained additional prestige with increased competition. Every spring, each computer-related department is asked to submit its best doctoral dissertation produced during the previous year. Each submission is examined by four external expert reviewers. Based upon these reviews, five dissertations are selected for the final round of competition. The final five dissertations are all reviewed by the each of the six members of the selection committee. This committee determines the award winner.

This year forty-four dissertations were submitted. The selection committee was composed of Doug DeGroot, Larry Dowdy, David Johnson, Michael Marcotty, Fred Maryanski, and Jack Minker. Unlike previous years, co-winners of the award were selected. This decision was based upon the unique and significant contributions of each. The co-winners are Ketan Mulmuley for his dissertation entitled "Full Abstraction and Semantic Equivalence" and Johan Håstad for his dissertation entitled "Computational Limitations for Small Depth Circuits" . Dr. Mulmuley's work was supervised by Professor Dana Scott at Carnegie-Mellon University. Dr. Håstad's work was supervised by Professor Shafi Goldwasser at the Massachusetts Institute of Technology.

This book presents Mulmuley's thesis. It provides important advances in the semantics of programming languages and in automated theorem proving. The thesis solves a major open problem in the field of denotational semantics, that of finding fully abstract models of the typed lambda calculus. Proving such existences are quite detailed and researchers have long expressed a need for a mechanized method to handle most of the details. Mulmuley provides and implements such a mechanized method. This method is a major step in constructing automated theorem provers for denotational semantics. Mulmuley's contributions to the field are truly outstanding.

Larry Dowdy, Chairman
ACM Doctoral Dissertation Award Subcommittee

Preface

This thesis studies the relationship between denotational and operational semantics of a language. There are two important problems which arise in this context: the problem of full abstraction and the problem of semantic equivalence. We are concerned here with the Scott-Strachey approach to programming language semantics. In this approach, a language is given semantics by mapping each language construct to its meaning in an appropriately constructed mathematical domain – this map is called a denotational semantics of the language. The map is denotational in the sense that the denotation of a complex term depends only on the denotations of its constituents and not on their structure. Of course, if the language is to be of any use at all, it must also have an operational semantics. Generally this operational semantics is specified in terms of a compiler or an interpreter.

The first question which arises is: are the denotational and operational semantics equivalent? This is the problem of semantic equivalence. Milne and Reynolds gave a general technique for proving such semantic equivalences. Their technique involves constructing certain predicates, called inclusive predicates, which connect the domains used for denotational and operational semantics. The most difficult part of their technique is showing the existence of such inclusive predicates. Unfortunately these existence proofs are known to be quite complicated, hence one is reluctant to carry out these proofs. Moreover, so far one did not know any nontrivial example of a system of equations over inclusive predicates which does not have a solution. In the absence of such a counterexample it does not seem justified to carry out the complicated existence proofs with all their details. In this thesis we shall construct such a counterexample through diagonalization. This means one must carry out the existence proofs with care. It has been suspected for some time that much simpler methods for proving these existences will be discovered. For example, several people suggested that there might be a language which can define most of the frequently arising predicates. Unfortunately the above mentioned counterexample shows that there are some fundamental difficulties in doing this. However, in this thesis we shall give a theory to prove such semantic equivalences which has the distinct advantage of being mechanizable. In fact, a system, which we shall call IPL (Inclusive Predicate Logic), was imple-

mented on top of LCF which can almost automatically prove the existence of most of the inclusive predicates which arise in practice.

Another issue this thesis deals with is the one of full abstraction. A model of a programming language is said to be fully abstract if denotations of two language constructs are equal whenever those constructs behave the same in all programming contexts and vice versa. For a special case of typed lambda calculus, PCF, it was shown by Plotkin that the classical model consisting of all continuous functions is not fully abstract. (See [Plotkin1]). However, he was able to make the model fully abstract by adding to the language a new programming construct which provided a *parallel or* facility. Milner showed that, under reasonable assumptions, typed lambda calculus has a unique fully abstract, extensional model, and he constructed the model syntactically. (See [Milner].) (A model is said to be extensional if any object of a higher type is uniquely determined by its action on the objects of an appropriate lower type.) In this thesis we shall connect Milner's model to the classical model. We shall show that one can construct an extensional, fully abstract and algebraic model of typed lambda calculus which is a *homomorphic retraction* (or a submodel) of the classical model, if the classical model is based on complete lattices. Milner's unique fully abstract model, which is based on consistently complete cpos instead of complete lattices, can be recovered from our fully abstract submodel in a very simple way. As a fully abstract model reflects the operational behaviour of the language precisely, our result says that the lattice theoretic model of a typed lambda calculus already contains within itself a submodel which precisely reflects the operational behaviour. This is particularly surprising as the lattice-theoretic models are generally constructed without giving much attention to the operational semantics of a language. Moreover, we shall show that the theory can extended to the case when a language has reflexive (i.e. recursively defined) types. This is important as most of the 'real' programming languages have recursively defined types. One distinguishing feature of this theory is that it uses the same inclusive predicates, which can be used to show the semantic equivalence between denotational and operational semantics, to construct fully abstract, extensional submodels. This strengthens our belief that the proofs of semantic equivalence and full abstraction go hand in hand.

The outline of the thesis is as follows.

In Chapter 1 we provide an introduction to Scott's theory of domains.

In Chapter 2 we introduce the problem of inclusive predicate existence.

We also present here the counterexamples using diagonalization and self application to show that this existence problem is indeed nontrivial.

In Chapter 3 we show how useful inclusive predicates can be, when they exist. Using certain inclusive predicates, we construct for typed lambda calculi fully abstract, extensional, algebraic submodels of their lattice theoretic models. We shall also see how Milner's unique fully abstract model, which is based on consistently complete cpos instead of lattices, can be recovered from our fully abstract submodel in a very simple way.

In Chapter 4 we extend the theory of Chapter 3 to the case when the language has reflexive types.

In Chapter 5 we give a mechanizable theory for proving the existence of inclusive predicates.

In Chapter 6 we describe the system IPL (Inclusive Predicate Logic) which mechanizes and automates the theory of Chapter 5.

Finally Chapter 7 lists some open problems and the directions for future research.

Acknowledgements

I am deeply grateful to my advisor Prof. Dana Scott who introduced me to the field of semantics and whose innumerable invaluable suggestions always kept me on a right track. Without his able guidance this thesis would simply not have resulted. Very very special thanks to Steve Brookes with whom I had several illuminating discussions. He went through the thesis very carefully and provided lots of useful suggestions. I also wish to thank Mike Gordon, Gordon Plotkin, Rick Statman and Glynn Winskel whose criticisms have been of a great value to me. A special mention must be made of Roberto Minio who was always willing to help me in typesetting the thesis.

Needless to say, I am most grateful to the CMU computer science department for providing such a wonderful research community and environment.

Full Abstraction and Semantic Equivalence

Contents

Chapter 1

Domain Theory

In the Scott-Strachey approach to programming language semantics a language is given a semantics by mapping each of its syntactic constructs into a set of mathematical domains. The hope is that one will be able to reason about programming language constructs by using the properties of these domains. Naturally the success of the approach depends upon how nice and convenient the mathematical properties of these domains are, and whether these domains are powerful and general enough to be used in giving semantics to a large class of programming languages. The domain theory of Scott is a theory of constructing domains meeting the above demands. We shall provide a brief overview of that theory in this chapter. All of the material in this chapter can be found in [Scott1].

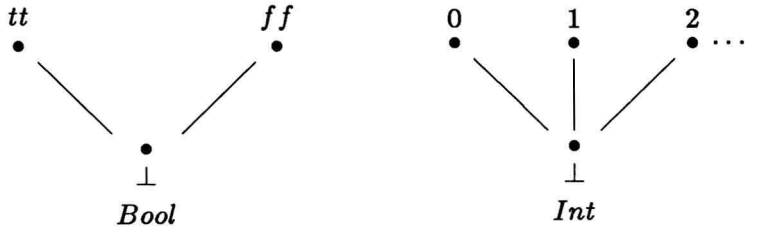
1.1 Complete Partial Order

The domains used in giving denotational semantics to programming languages are in technical language complete partial orders. We shall introduce in this section the notion of a complete partial order.

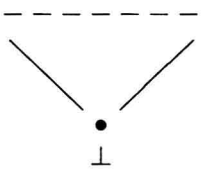
By a partial order we shall mean a transitive, reflexive, antisymmetric relation. A partially ordered set S is called directed if, for any $x, y \in S$, there exists a $z \in S$ such that $x \sqsubseteq z$ and $y \sqsubseteq z$, i.e., any two elements of S have an upper bound in S.

A partially order set D is called a complete partial order (cpo) if it has a least element, which we shall call $\perp$, and any directed subset of D has a least upper bound (lub) in D. Note that least elements and lubs are unique.

Rather trivial examples of cpos are *Bool* and *Int* as shown below. (In the diagrams we assume that the ordering 'increases' in the upward direction.)



For a more nontrivial example, consider any set X. Then $P(X)$, the powerset of X, is a cpo under the usual subset relationship ordering. A cpo should be visualized as follows:



1.2 Mappings Between CPOs

Given two cpos C and D, a function f from C to D is called monotonic if, whenever $x \sqsubseteq y$ in C, $f(x) \sqsubseteq f(y)$ in D.

A monotone function f is called continuous if, for any directed subset S of C, $f(lub(S)) = lub(f(S))$. (Note that monotonicity of f implies that $f(S)$ is directed whenever S is.)

As cpos are closed under the *lub* operation on directed sets, a continuous function is the most natural notion of a morphism on cpos. Technically, if we take cpos as objects and continuous functions on them as morphisms, we get a category which we shall call the category of cpos. (For an elementary introduction to category theory see [Arbib].)

If C and D are cpos then we shall denote by $C \to D$ the set of continous functions from C to D. Under the pointwise ordering $C \to D$ is a cpo if C and D are. We shall see later how we can actually regard $\to$ as a functor on the category of cpos.

Given a continous function f from D to D, $z \in D$ is called a fixpoint of f if $f(z) = z$.