Fully Abstract Models of Programming Languages*

Allen Stoughton

School of Mathematical and Physical Sciences
University of Sussex

*The preface and first chapter of a monograph that was published in 1988 in Pitman's *Research Notes in Theoretical Computer Science* series.

Preface

This monograph presents a language-independent theory of *fully abstract* denotational semantics of programming languages—models that identify program fragments exactly when they are operationally interchangeable—and uses this theory to show the existence or nonexistence of such models for several example programming languages. It is intended for researchers in programming language semantics, and is mathematically self-contained: only naive set theory and some very basic notions of category theory are assumed. Some familiarity with universal algebra and domain theory would be helpful, however.

This monograph is a revision of the author's Ph.D. thesis [Stou], which was submitted to the University of Edinburgh in August, 1986. It is a pleasure to acknowledge many stimulating discussions with my thesis supervisor, Gordon Plotkin. Numerous conversations with Marek Bednarczyk, Ilaria Castellani, Peter Dybjer, Matthew Hennessy, Kim Larsen, David McCarty, Robin Milner, Peter Mosses, Andrew Pitts, K.V.S. Prasad, Edmund Robinson, Dave Schmidt, Oliver Schoet and Colin Stirling, as well as particular talks with Gérard Berry, Evelyn Nelson and Glynn Winskel, were also helpful. Special thanks are due to Gérard Berry, Peter Dybjer and David McCarty for critically reading my thesis, and to Gérard Berry, Rance Cleaveland and Andrew Pitts for their comments on a draft of this monograph.

I was financially supported by a University of Edinburgh Studentship and research fellowships from the Department of Computer Sciences of Chalmers University of Technology and the Science and Engineering Research Council of Great Britain.

Contents

1 Introduction

1.1 Program Equivalence and Full Abstraction

Notions of program equivalence are fundamental to the theory and practice of programming languages. They are the semantic basis for program optimization and can be used to justify the correctness-preserving transformations that are employed by program manipulation systems. Notions of program equivalence are generally substitutive in the sense that the results of embedding equivalent terms (program fragments) into a context (a term with "holes" in it) are also equivalent. Thus a programmer can replace fragments of a program by equivalent terms without considering the details of the whole program.

Program equivalences are typically defined according to the following paradigm. Terms that are considered to be directly executable and observable are designated as *programs*, and their behaviour is defined. Then two terms are defined to be equivalent if and only if (iff) they have the same behaviour in all program contexts, i.e., iff one can be replaced by the other in any program without affecting the behaviour of that program. Thus term equivalence is reduced to program behaviour.

The distinction between terms and programs is often suggested by the syntactic categories of programming languages. For example, in an imperative language with statements and expressions the statements might be taken to be the programs, reflecting the view that expressions can only be executed as parts of statements. For languages with block structure, i.e., in which identifiers can be statically bound, it is common to take the closed terms as the programs.

By the behaviour of programs we mean the actions of programs that are visible to external observers. Program behaviours for a deterministic programming language might simply be functions from inputs to outputs, whereas behaviours for languages with communicating processes might consist of communication histories. Much depends upon the level of detail that external observers are allowed to see.

It is also possible to consider notions of program ordering, i.e., notions of when one term should be considered less defined, or convergent, than another. Program orderings are typically defined by ordering the set of program behaviours and then defining one term to be less than another iff the behaviour of the first is less than that of the second in all program contexts.

Program behaviours and their orderings can be defined as abstractions of both operational and denotational semantics, although the literature is currently biased toward the use of operational semantics. Examples of the use of denotational semantics in this way are given in this monograph. Often there are multiple natural notions of behaviour that can be defined via a given semantics. Examples of behaviourally defined program orderings and equivalences can be found in [Mil1], [Mil3], [Plo1] and [HenPlo1].

Once a notion of program equivalence has been selected for a programming language, its properties must be determined and proof techniques found. Denotational semantics, as developed by Scott, Strachey and their followers (see [Stoy] for an introduction and extensive references), is a suitable framework for these activities. The idea is to reduce the equivalence of terms to the equality of their semantic values in appropriate models, i.e., to semantically capture the notion of program equivalence. Thus it is necessary to work with models that are *equationally correct* (or simply *correct*) in the sense that only equivalent terms are identified (mapped to the same semantic value). Models with the ideal property that exactly the equivalent terms are identified are called *equationally fully abstract* (or simply *fully abstract*).

Similarly, one can judge denotational semantics with reference to notions of program ordering. A model is said to be *inequationally correct* with reference to a program ordering iff one term is less than another in the program ordering whenever the meaning of the first is less than that of the second in the model, and *inequationally fully abstract* iff one term is less than another in the program ordering exactly when the meaning of the first is less than that of the second in the model.

For models to be useful for reasoning about program equivalences or orderings, it is necessary that their structure be understandable independently from those equivalences or orderings; informally, we call such models *natural*. For example, models synthesized using the standard constructions of denotational semantics are generally natural, in contrast to term models, i.e., models constructed from equivalence classes of terms, etc.

The idea of judging denotational semantics with reference to predefined notions of program ordering and equivalence is due to Milner [Mil1] and Plotkin [Plo1] and has been studied, for a variety of programming languages, by Abramsky, Berry, Curien, Hennessy and others. Research on full abstraction can be divided into two categories:

(i) The synthesis and analysis of natural models.

(ii) The theoretical study of the conditions under which fully abstract models

exist.
We consider each of these in turn.

For many programming languages, the standard techniques of denotational semantics yield natural models that are too concrete, i.e., correct but not fully abstract. Many common language features, such as functions of higher type, concurrency, storage allocation and data abstraction, are problematic. This phenomenon was first noticed in connection with a simple applicative programming language, based upon the typed lambda calculus, called PCF (Programming Computable Functions). Plotkin [Plo1] showed that the natural continuous function model of PCF is correct, but not fully abstract, with reference to its standard notion of program equivalence, which is based upon the total evaluation of closed ground terms. This lack of full abstraction is due to the presence of certain "parallel" elements in the model, which are not realized by terms in the programming language. In fact, Plotkin showed that if a "parallel conditional" is added to the language then the continuous function model of this extended language is fully abstract. The problem of finding a natural fully abstract model of the original language is still open, although much progress has been made by Berry, Curien and Winskel [BerCurLév]. A byproduct of their work is the *sequential algorithms* model of PCF [BerCur], which is fully abstract with reference to an alternative notion of program equivalence that is sensitive to the order and extent of evaluation of function arguments. See [BerCurLév] for an excellent survey of this and other research into the full abstraction problem for PCF.

Other examples of the search for natural fully abstract models can be found in [HenPlo1], which considers a simple parallel programming language, [Abr1] and [Abr2], which treat a nondeterministic applicative language with infinite streams, and [Bro] and [HalMeyTra], which deal with Algol-like languages. Many open problems exist.

The difficulty of finding natural fully abstract models for many programming languages has led to the theoretical study of the conditions under which fully abstract models exist. Proofs of the existence or nonexistence of fully abstract models of programming languages are relative, of course, to what count as models of those languages. Positive results spur on the search for natural models, whereas negative ones indicate that the class of models being considered must be widened.

The study of the existence of fully abstract models can be carried out within the framework of initial algebra semantics [Sco][ADJ1][CouNiv]. Programming language syntax is specified in this framework by many-sorted signatures, whose sorts and operators correspond to the syntactic categories and constructs, respectively, of programming languages, and models are universal algebras whose carriers have certain order-theoretic structure and whose operations preserve that structure. Usually the carriers are taken to be complete partial orders (cpo's) and the operations continuous functions, but it is also possible to work with weaker notions of continuity [AptPlo][Plo2] or to generalize from partial orders to categories [Leh][Abr2]. The meanings assigned by models to iteration and recursion constructs are normally required to be least fixed points of appropriate unary derived operations. For example, the meaning of a while-loop *while E do S od* should be the least fixed point of *if E then S; $-$ else skip fi*. Many additional requirements may be set for models of particular programming languages, e.g., extensionality for models of applicative languages.

Positive results are typically proved via term model constructions. Such techniques were first used by Milner, who constructed a fully abstract model of the combinatory logic version of PCF [Mil2]. His construction was simplified and applied to the typed lambda calculus version of PCF by Berry [Ber1]. Similar techniques were used by Hennessy and Plotkin to construct fully abstract models of two variants of CCS [HenPlo2][Hen]. These term model constructions proceed, roughly, as follows. One designates certain terms as "semantically finite", orders them by the language's notion of program ordering, and then makes them into an ω-algebraic cpo, using the familiar ideal completion. An algebra is then defined using this cpo as its carrier. In Berry's construction, the syntactic projections $\Psi_n M$ of arbitrary terms M are taken as the semantically finite terms.

Recently, Mulmuley has considered the problem of connecting the continuous function model of the combinatory logic version of PCF with Milner's fully abstract model [Mul]. Using operationally defined inclusive predicates, he defines a fully abstract model as a retract of the complete lattices version of the continuous function model and then removes the top elements from this model, thus yielding Milner's model. In this ingenious construction the unwanted, parallel functions are retracted either to wanted, sequential ones or to the top elements. Thus the technique is not applicable to the usual continuous function model, which is based instead upon cpo's; since the complete lattices model is less abstract than the cpo model [Plo1], this may be seen as a disadvantage. In addition, Mulmuley's retraction does not preserve function application, and thus is not a homomorphism of algebras, and his retract model is not a combinatory algebra, because of the interaction between the K combinators and the top elements.

The first negative result was proved by Apt and Plotkin [AptPlo] for a nondeterministic imperative programming language with random assignment, i.e., the facility for choosing an arbitrary natural number and assigning it to a variable. They prove that there does not exist a fully abstract model that is based upon cpo's and continuous functions for this language. This is because there are programs (such as the one that chooses a natural number and then decrements it until it becomes zero) that always terminate, but whose finite approximations all have the possibility of divergence. They are able, however, to give a natural fully abstract model that is based upon a weaker notion of continuity. Abramsky, following this work, has proved a similar negative result for a nondeterministic applicative programming language with infinite streams [Abr3].

1.2 A Theory of Fully Abstract Models

All of the research described above has focused on full abstraction for specific programming languages. In this monograph we try to develop a theory of fully abstract models of programming languages that is applicable to programming languages in general. The goal is to develop a unified framework in which simpler proofs of the existing positive and negative results can be given and new results can be proved. The following paragraphs summarize the contents of the monograph.

We begin by building a mathematical framework for studying full abstraction, based upon initial algebra semantics. As models we take *complete ordered algebras*, i.e., many-sorted universal algebras whose carriers are sort-indexed families of complete partial orders and operations are continuous functions. Following [CouNiv], every signature is required to contain a distinguished nullary operator Ω of each sort, which stands for divergence or nontermination, and is interpreted as the least element of its sort in every model. Although programming languages rarely contain such constants explicitly, many languages for which divergence is possible in all syntactic categories do contain terms that the constants Ω can be modelled after, e.g., *while true do skip od*, in some imperative languages. **Chapter 2** consists of the definitions and theorems concerning universal algebras and ordered algebras that will be needed in the sequel. In particular, we prove several quotienting and completion theorems that will be used in term model constructions.

Chapter 3 is devoted to the definitions and elementary properties of full abstraction and least fixed point models. We consider three kinds of full abstraction (and also correctness): *equational*, *inequational* and *contextual*. The first two are as described above, and the third is the natural generalization of equational full abstraction from ordinary terms to contexts. Formally, notions of program equivalence are congruences over the term algebra, and notions of program ordering are substitutive pre-orderings over the term algebra in which the maximally divergent terms Ω are least elements. Least fixed point models are intended to assign iteration and recursion constructs meanings that are least fixed points of appropriate unary derived operations. Such requirements are formally expressed in our framework by *families of least fixed point constraints*, which specify that the meanings of certain terms should be least upper bounds of the meanings of certain directed subsets of the ordered term algebra. We also consider contextually least fixed point models, which are the natural generalization of least fixed point models from terms to contexts.

In **chapter 4** we study two programming languages within our framework. The first is the combinatory logic version of PCF, and the second is an imperative language with explicit storage allocation and higher and recursive types, which we call TIE. We give denotational semantics for both of these languages, define notions of program ordering and equivalence as abstractions of these models, in a uniform manner, and show that the models are inequationally correct with reference to these notions of ordering. In contrast, the model of PCF is already known not to be fully abstract, and we conjecture that neither is our model of the second language.

In **chapter 5** we give necessary and sufficient conditions for the existence of correct and fully abstract models, for each of the three kinds of correctness and full abstraction. The condition for the existence of inequationally fully abstract models is the cornerstone of these results. An inequationally fully abstract model of a programming language exists iff its notion of program ordering satisfies the constraints in the closure—under the operations of the term algebra—of its family of least fixed point constraints. Showing the necessity of this condition is straightforward. Its sufficiency is proved via a term model construction: the ordered term algebra is quotiented by the notion of program ordering, and then embedded into a complete ordered algebra in a way that preserves the least upper bounds corresponding to the constraints in the closure of the family of least fixed point constraints. The chapter concludes with several theorems concerning the existence of initial objects and the nonexistence of terminal objects in various categories of models.

Chapter 6 consists of simplified proofs of the negative results of [AptPlo] and [Abr3], using the condition for the existence of equationally fully abstract models given in chapter 5. Although our theory is directly applicable to Abramsky's nondeterministic applicative language with streams, we prefer to work instead with a nondeterministic imperative language with infinite output streams. Since the streams of our language are unreadable, in contrast to those of Abramsky's language, we achieve a slight sharpening of his result. The notions of program equivalence for the languages of this chapter are defined via operational semantics, and no model-theoretic reasoning is used in the proofs of the negative results.

In **chapter 7**, we investigate two approaches to obtaining fully abstract models from correct ones. In the first, we use the condition for the existence of inequationally fully abstract models given in chapter 5 in order to develop useful necessary and sufficient conditions involving the existence of correct models. In the second, we consider the possibility of collapsing correct models, via continuous homomorphisms, to fully abstract ones. We show that this is not always possible—indeed the natural continuous function model of PCF provides a counterexample—but give sufficient conditions for its possibility. Both of these approaches yield fully abstract models for the languages introduced in chapter 4, and, more generally, for languages whose notions of program ordering and equivalence are defined as abstractions of models using the technique of chapter 4. In the case of PCF, we are able to continuously collapse the reachable inductive subalgebra of the continuous function model to Milner's fully abstract model, thus providing a pleasing, algebraic solution to Mulmuley's problem of relating these models.

Finally, in **chapter 8**, we consider the limitations of the monograph and the corresponding possibilities for further research.

Bibliography

[ADJ1] J. Goguen, J. Thatcher, E. Wagner and J. Wright. Initial algebra semantics and continuous algebras. *J. ACM* 24, 1, 1977.

[ADJ2] J. Wright, E. Wagner and J. Thatcher. A uniform approach to inductive posets and inductive closure. *Theoretical Computer Science* 7, 1978.

[Abr1] S. Abramsky. Experiments, powerdomains and fully abstract models for applicative multiprogramming. *Proc. Conference on Foundations of Computation Theory*, Lecture Notes in Computer Science, vol. 158, Springer-Verlag, 1983.

[Abr2] S. Abramsky. On semantic foundations for applicative multiprogramming. *Proc. 10th ICALP*, Lecture Notes in Computer Science, vol. 154, Springer-Verlag, 1983.

[Abr3] S. Abramsky. Semantic foundations of applicative multiprogramming. *Proc. Workshop on Semantics of Programming Languages*, Programming Methodology Group, University of Göteborg and Chalmers University of Technology, 1983.

[AptPlo] K. Apt and G. Plotkin. Countable nondeterminism and random assignment. *J. ACM*, to appear.

[Ber1] G. Berry. *Modèles complètement adéquats et stables des lambda-calculs typés.* Thèse de Doctorat d'Etat, Université Paris VII, 1979.

[Ber2] G. Berry. *Some syntactic and categorical constructions of lambda-calculus models.* Technical report No. 80, INRIA, 1981.

[BerCur] G. Berry and P.-L. Curien. Sequential algorithms on concrete data structures. *Theoretical Computer Science* 20, 1982.

[BerCurLév] G. Berry, P.-L. Curien and J.-J. Lévy. Full abstraction for sequential languages: the state of the art. In M. Nivat and J. Reynolds (editors), *Algebraic Methods in Semantics*, Cambridge University Press, 1985.

[Bro] S. Brookes. A fully abstract semantics and a proof system for an ALGOL-like language with sharing. *Proc. Conference on Mathematical Foundations of Programming Semantics*, Kansas State University, 1985.

[CouNiv] B. Courcelle and M. Nivat. Algebraic families of interpretations. *Proc. 17th IEEE Symposium on Foundations of Computer Science*, 1976.

[CouRao] B. Courcelle and J.-C. Raoult. Completions of ordered magmas. *Fundamenta Informaticae* 3, 1, 1980.

[HalMeyTra] J. Halpern, A. Meyer and B. Trakhtenbrot. The semantics of local storage, or What makes the free-list free? *Proc. 11th ACM Symposium on Principles of Programming Languages*, 1984.

[Hen] M. Hennessy. A term model for synchronous processes. *Information and Control* 51, 1, 1981.

[HenPlo1] M. Hennessy and G. Plotkin. Full abstraction for a simple parallel programming language. *Proc. 8th MFCS*, Lecture Notes in Computer Science, vol. 74, Springer-Verlag, 1979.

[HenPlo2] M. Hennessy and G. Plotkin. A term model for CCS. *Proc. 9th MFCS*, Lecture Notes in Computer Science, vol. 88, Springer-Verlag, 1980.

[Grä] G. Grätzer. *Universal Algebra*. Springer-Verlag, 1979.

[Leh] D. Lehmann. Categories for fixed point semantics. *Proc. 17th IEEE Symposium on Foundations of Computer Science*, 1976.

[Mar] G. Markowsky. Chain-complete posets and directed sets with applications. *Algebra Univ.* 6, 1976.

[Mil1] R. Milner. Processes: a mathematical model of computing agents. *Proc. Logic Colloquium '73*, North Holland, 1975.

[Mil2] R. Milner. Fully abstract models of typed λ-calculi. *Theoretical Computer Science* 4, 1977.

[Mil3] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag, 1980.

[Mul] K. Mulmuley. *Full abstraction and semantic equivalence.* The MIT Press, 1987.

[Nel] E. Nelson. Z-continuous algebras. *Continuous Lattices*, Lecture Notes in Mathematics, vol. 871, Springer-Verlag, 1981.

[Plo1] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science* 5, 1977.

[Plo2] G. Plotkin. A powerdomain for countable nondeterminism. *Proc. 9th ICALP*, Lecture Notes in Computer Science, vol. 140, Springer-Verlag, 1982.

[Plo3] G. Plotkin. *Domains.* Lecture Notes, Department of Computer Science, University of Edinburgh, 1980.

[Sco] D. Scott. The lattice of flow diagrams. *Proc. Symposium on Semantics of Algorithmic Languages*, Lecture Notes in Mathematics, vol. 188, Springer-Verlag, 1971.

[SmyPlo] M. Smyth and G. Plotkin. The category-theoretic solution of recursive domain equations. *SIAM J. Computing* 11, 4, 1982.

[Stou] A. Stoughton. *Fully Abstract Models of Programming Languages.* Ph.D. Thesis, Technical Report CST–40–86, Computer Science Department, University of Edinburgh, 1986.

[Stoy] J. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory.* The MIT Press, 1977.

Index of Definitions