Introduction to the Literature on Semantics

Gary T. Leavens

Iowa State University

Follow this and additional works at: http://lib.dr.iastate.edu/cs_techreports

Part of the Computer Sciences Commons

Recommended Citation

http://lib.dr.iastate.edu/cs_techreports/96

This Article is brought to you for free and open access by Digital Repository @ Iowa State University. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact hinefuku@iastate.edu.
Introduction to the Literature
On Semantics

Gary T. Leavens

TR #94-15
August 1994

Keywords: programming languages, axiomatic semantics, denotational semantics, operational semantics, type systems, polymorphism, type theory, data abstraction.


© Gary T. Leavens. Permission is granted for you to make copies for educational and scholarly purposes, but not for direct commercial advantage, provided this notice appears on all copies. All other rights reserved.

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1040, USA
Introduction to the Literature on Semantics

Gary T. Leavens
Department of Computer Science, Iowa State University
Ames, IA, 50011-1040 USA
leavens@cs.iastate.edu
August 16, 1994

Abstract

An introduction to the literature on semantics. Included are pointers to the literature on
axiomatic semantics, denotational semantics, operational semantics, and type theory.

The following is an selective introduction to the literature on programming language semantics.
Since this is an introduction, only a small fraction of the literature can be mentioned here. References are included because they have intrinsic interest, although some are included simply because they are the original sources and are likely to be referenced by others doing related work (e.g. [Chu41]). If you wish to probe an area more deeply, you might start with the papers mentioned, follow their references, and also use the Science Citation Index to see what papers have referenced the ones mentioned.

1 General Sources

Two recent texts that cover many different approaches to semantics are: [NN92] [Win93]. Articles discussing principal topics in semantics, containing many references, appear in volume B of the Handbook of Theoretical Computer Science [vL90].

Journals that include a substantial coverage of semantics include ACM Transactions on Programming Languages and Systems (TOPLAS), ACM SIGPLAN Notices, IEEE Transactions on Software Engineering, Information and Computation (formerly Information and Control, Academic Press), and Acta Informatica (Springer-Verlag). Related areas, such as specification and verification have their own journals (e.g., ACM Transactions on Software Engineering and Methodology, Science of Computer Programming, and Formal Aspects of Computing).

Conferences devoted to topics related to semantics include the IEEE Annual Symposium on Logic in Computer Science (LICS), the Annual ACM Symposium on Principles of Programming Languages (POPL), Mathematical Foundations of Programming Semantics (MFPS), the European Symposium on Programming (ESOP), the ACM Symposium on LISP and Functional Programming (LFP), the International Colloquium on Automata, Languages, and Programming (ICALP), and the International Symposium on Mathematical Foundations of Computer Science. Conferences not sponsored by the ACM or IEEE are often available in the Springer-Verlag series of Lecture Notes in Computer Science (LNCS), and workshops are often available in the series Workshops in Computing. The above list does not include conferences devoted to software engineering (e.g., TAPSOFT) or particular methods (e.g., category theory) where some important work is also found.

1
2 Mathematical Background

The basics of universal algebra underly much of the work in semantics. Gratzer’s monumental book [Gra79] is a standard reference. A generalized notion of homomorphism that is often used in semantics is Statman’s logical relations [Sta85], explained in [Mit90].

Another foundation is the area of mathematical logic. A classic approach, using proof theory and model theory is [End72]. More recent approaches, favoring proof theory (especially predicate logic) at the expense of model theory and mathematical depth and emphasizing calculation, include [DS90] [GS94] [MW93].

2.1 λ-Calculus

A standard reference on Church’s (untyped) λ-calculus [Chu41] is Barendregt’s book [Bar84]. This book is at times highly technical, but also has sensible definitions and introductory material. A shorter account, with a reasonable introduction to the λ-calculus is found in [Bar90]. Other introductions include [Gor88] [Sch94, Sections 6.1–6.2].

The use of λ-calculus for describing programming languages and as the inspiration for programming language design has been investigated by Landin [Lan65] [Lan66] and many others.

For the typed λ-calculus, a standard reference is [GLT89]. A good introduction can be found in [Gun92, Chapter 2] and [Sch94, Sections 6.4]; both of these texts discuss the semantics of the typed lambda calculus in detail. Schmidt’s book [Sch94] is also a good introduction to higher-order typed lambda calculi, and contains references to work on such higher-order extensions.

2.2 Category Theory

Increasingly, especially in denotational semantics, category theory [Lan71] [LS91] is used in research and for the elegant presentation of results. For example, categorical logic and the semantics of the typed lambda calculus are discussed in [LS86].

For me, the best introduction seems to be the first 5 chapters of [Gol84]. There are now several articles [Hoa89] and books [BW90] [AL91b] [Pie91] [Wal91] explaining category theory to computer scientists. For those familiar with (or needing) background in the semantics of abstract data type specifications, a good introduction is Burstall and Goguen’s paper [BG82].

3 Axiomatic Semantics

Hoare pointed out, in his seminal paper [Hoa69, Section 6], that one could define a programming language by insisting “that all implementations of the language shall ‘satisfy’ the axioms and rules of inference which underlie proofs of the properties of the programs expressed in the language.” (However, Hoare attributes this idea to Floyd [Flo67].) Such a semantics is useful because it directly aims to support program verification.

Because a logic for program verification can be used to define a programming language, it is difficult to disentangle work on program verification from work on semantics. Even if program verification is not itself a form of semantics (rather, it is an application), it still provides important motivation for axiomatic semantics. Therefore, verification is also discussed in this section.

3.1 Program Verification

Program verification is concerned with showing that a particular program meets its specification. This may be accomplished in several ways; for example, reasoning from the denotational semantics
of the program. However, the term “program verification” is usually implies a syntactic proof in a specially constructed logic.

Floyd introduced the inductive assertions and well-founded sets methods for program verification in 1967 [Flo67] (see also [LS87, Chapter 8]). (See [Cou90] for historical references to the work of Naur, Von Neumann, and Turing, who anticipated this work.) The term inductive assertions refers to the assertions that are put at every branch point in the program’s flow graph; these give rise to verification conditions, which state that along every path, the correctness of the assertions is preserved. This shows partial correctness (that the program is correct if it terminates). The well-founded sets method allows proofs of termination.

Hoare [Hoa69] was the first to present verification in terms of a logic for partial correctness that was compositional, so that program proofs are done by induction on the syntax of programs. This paper sparked a wealth of research in program verification. Some surveys of results of this research are [Apt81], [FB86], and [Cou90], the latter of which also cites other surveys. A technical reference is [LS87, Chapter 9].

A very introductory tutorial on verification from the software engineering perspective is [LG86, chapter 11]. The idea of developing a proof of a program at the same time the program is being developed has been eloquently advocated by Dijkstra and Gries [Dij76] [Gri81]. See [Gor88] for a more theoretical introduction that treats some aspects of theorem proving.

Concurrency has always been a prime application area, because concurrent programs are so difficult to debug. Verification of concurrent programs may be handled with the Owicki-Gries method [OG76]. Other techniques use temporal logic (e.g., [OL82]) and “transition axioms” [Lam89]. An introduction to Hoare-style verification that discusses concurrent and distributed programs is [Al91a].

3.1.1 Abstract Data Types

Hoare also was the first to consider how verification and abstract data types interact [Hoa72]. His technique allows one to separately verify the correctness of a type’s implementation and programs that use that type, and is based on the use of representation invariants and abstraction functions. A practical implementation of these ideas is found in [NIIN80]. VDM also uses these techniques [Jon90]. More recent work in this area is found in [Sch90] and [Nip89].

Abstract data types (ADTs) can be specified algebraically [GHM78] [EM85] or by using pre- and post-conditions [BJ82] [Jon86] [Win87]. See [LG86, chapter 11] for a tutorial on program verification using abstract data types.

A related aspect is how the structure of a proof of correctness may help follow the hierarchical (i.e., layered) structure of the implementation design [GMP89] [Sch82].

3.1.2 Transformational Programming

Hoare’s original paper [Hoa69], treated verification as something to be done after a program was written. However, Hoare [Hoa71] and others were soon advocating the development of proofs at the same time as programs. Dijkstra and Gries became prime advocates of this technique [Dij76] [Gri81]. More recent treatments in this style advocate a calculational approach [Coh90] [GS94]. One formalization of this approach is the refinement calculus [Bv89a][Bv89b] [Mor90] [MG90] [MV94].

The idea of formal development of program and proof has found favor in the functional programming community as well. The idea behind transformational programming is to transform the specification of a program into an efficient version [BD77] [MW80] [Bal81] [CHT81]; these references are characterized by their work with equational logic (instead of predicate transformers or a Hoare
logic), their use of algebraic techniques, and the general focus on functional languages. Some recent work in this area can be found in [Mee87] [Bir89a] [Bir89b] [Tal88]. Transformational (or algebraic) techniques also underly the work on Backus’s FP [Bac78] and Hoare’s “Laws of Programming” for CSP [HHJ+87].

Contrary to popular belief, it is also possible to reason “equationally” about programs with side-effects and even continuations [Boe85] [FF86] [FI89].

3.2 Axiomatic Semantics of Programming Languages

A classic text on the semantics of programming languages which treats axiomatic semantics is [dB80a]. Predicte transformer semantics, using the “weakest precondition” operator were advocated by Dijkstra for the definition of his guarded command language [Dij75]. A monograph on predicate transformer semantics is [DS90] which goes over notational issues as well as the underlying mathematics. A recent monograph that updates [dB80a] using predicate transformers is [Hes92].

An example of the use of the axiomatic semantics to describe a programming language is Hoare and Wirth’s axiomatic definition of Pascal [HW73].

The use of an axiomatic semantics as a metric for the “goodness” of a language led to two language designs in the 1970s. Although never implemented fully, Alphard [SWL77] [Sha81] was interesting, and focused on support for both data abstraction and verification. The language Euclid [LGH78] [PHL77] was implemented, and has been used for various large projects.

3.3 Algebraic Semantics

While most axiomatic approaches use Hoare logic or variations on Hoare logic, there have been a few attempts to use the techniques of algebra and equational specification to define programming languages. Examples include [MA86] [BWP87].

4 Denotational Semantics

In contrast to axiomatic semantics, denotational semantics [Str66] [SS71] [MS76] [Sto77] [Sco81] [Sch86] explicitly constructs mathematical models of programming languages. A short summary of the denotational approach can be found in Tennent’s article “The Denotational Semantics of Programming Languages” [Ten76]. A recent survey is found in [Mos90].

Introductory texts include [Gor79], [All86], [Wat91], [Sch94]. An excellent new graduate text with more mathematical depth is [Gun92]. Standard works on denotational semantics are the books by Stoy [Sto77] and Schmidt [Sch86], both of which offer a comprehensive and mathematical treatment. Schmidt’s book [Sch86] has an excellent discussion of domain theory (see also [GS90]) and can be consulted for references to denotational descriptions of real languages.

One can use a typed functional programming language, such as Standard ML, to implement a denotational semantics. Two descriptions of this idea are [Wat86] [MA89].

Action semantics, an offshoot of denotational semantics, is described in [Wat91] and more fully in [Mos92].

Schmidt’s book [Sch86] also has a discussion of denotational semantics for concurrent systems. Hennessy’s book has a discussion of more recent work in this area [Hen88]. However, denotational approaches to concurrency have not been as successful as operational semantics.
5 Operational Semantics

A compiler or interpreter gives an operational semantics to a programming language. This style of semantics dates back to the earliest programming languages, and was first formalized in the idea of a meta-circular interpreter for LISP [McC60].

Meta-circular interpreters are still useful for teaching purposes, and for prototyping programming language designs. Several meta-circular interpreters for variants of LISP are discussed in Steele and Sussman’s paper The Art of the Interpreter [SS78]. An excellent and more readily accessible discussion is found in Abelson and Sussman’s book [ASS85], which uses Scheme. A more detailed treatment of interpreters is found in [Kam90] [FWH92]. See [KdRB91] for an approach using the CLOS meta-object protocol.

However, for mathematical convenience, one wants something more abstract than an interpreter or compiler. Landin overcame the circularity problem by the use of an abstract machine called the “SECD machine” [Lan64] (see also [Hen80]).

A more systematic style of operational semantics based on rewrite rules is found in Plotkin’s terminal transition systems [Plo77] also known as “structural operational semantics” [Plo81] [Hen90] or a “labeled transition system” [Ast91]. Hennessy’s book is an elementary introduction [Hen90]. This style of semantics has the advantage that it extends naturally to studies of concurrency [Mil90]. A classic reference for the operational semantics of concurrent processes is Milner’s book on CCS [Mil80]. See [Hen88] and [Mil90] for more recent work in this area.

6 Type Systems

Much of the modern study of semantics concerns type systems, which describe many aspects of the static semantics of programming languages. In contrast, axiomatic, denotational, and operational semantics describe the dynamic semantics of programming languages.

The purpose of type checking is nicely summarized by Morris as a mechanism that allows program modules to protect objects from unwanted discovery, modification, and impersonation [Mor73]. Wegbreit’s discussion of the extensible language EL1, is also good background [Weg74].

Schmidt’s recent book [Sch94] starts treating type systems in the context of programming language semantics. It also goes deeply into more advanced topics such as higher-order type systems and predicate-logic typing.

6.1 Polymorphism

An introductory survey of modern polymorphic type systems and research results is Cardelli and Wegner’s paper “On Understanding Types, Data Abstraction and Polymorphism” [CW85]. See also [Har84] [Car91] [DT88]; the latter two have much material related to object-oriented programming. A still more recent survey is [Mit90].

Standard references include Girard’s system $F_\omega$ [Gir71] (see also [Gir86] [GLT89]), and Reynolds independent work [Rey74], sometimes called the Girard-Reynolds second order lambda calculus (or SOL). Modern expositions are found in [MP85] [MH88] [Rey85] [Mit90] [Car91].

Other kinds of type information may be incorporated into a type system and checked at the same time as types [GL86] [LG88] [OG89].
6.2 Type Theory

Type theory, narrowly defined, uses the tools of constructive logic to study polymorphic type systems. A good introduction can be found in [Sch94, Chapters 8–10]. Logical inference systems can often be translated directly into type systems due to the “Formula as Types” notion or the “Curry-Howard isomorphism” [How80] [GLT89, Chapter 3] [Con89]. Thus much research in type theory lies on the border of mathematics and computer science. Another motivation is to use type information to capture behavioral specifications, thus allowing one to reason about programs in the programming language [NP83] [Dyb90].

The principal groups working on type theory include deBruijn and others working on AUTOMATH [dB80b], Martin-Löf’s and followers [ML75] [ML82] [Bac89], Constable’s group at Cornell has been very active in this area. Their language PRL which uses constructive mathematics, is described in the paper “Proofs as Programs” [BC85]. A related language is PL/CV3 [CZ84]. Coquand and Huet’s group is responsible for the “calculus of constructions” [CH88]. A recent survey with more references is [BH90a].

6.3 Data Abstraction: ADTs and OOP

Reynolds [Rey75] [Rey78] (see also [Coo91]) distinguished two ways in which a type system could support data abstraction. The first way is to have the language give an object different types outside and inside a defining module; this Reynolds called “user defined types”. User defined types also go by the name of “abstract data types” (ADTs), and their support in a type system is exemplified by the language CLU [LSAS77] [LG86].

Mitchell and Plotkin connected this approach, ADTs, to the second-order lambda calculus [MP85]. This idea has been used to show some representation independence results [Mit86]. For other work along these lines, see for example, [Mac86] [CL90] [Mit90].

The second approach to supporting data abstraction Reynolds called “procedural data abstraction”. Today it goes under the more common name of “object-oriented programming” (OOP) [DMN70] [BDMN73] [GR83]. Some languages exemplifying typed support for OOP include SIMULA [BDMN73], and Eiffel [Mey88] [Mey92], both of which have insecure type systems [Coo89]. The language Trellis/Owl [SCB+86] features by-name type checking and subtyping. By contrast, Emerald [BHJ+86] [BHJ+87] [BH90b] [BH91] and Quest [Car91] feature structural subtyping.

Some seminal references on types and OOP are reprinted in [GM94]. Recent theoretical work on types for OOP includes the following [Car88b] [Car88a] [CM89] [AC90] [CMMS91] [Car93] [CCH+89] [CHC90] [Coo89] [BTGS90] [HP90] [MMM91] [BCM+93] [BL90] [BM92] [Aba93] [Bru93]. (Cardelli is one of the most active in this area, and most of the literature will cite one of his papers.) For work that directly bears on multimethods (as in CLOS), see [Rey80] [Ghe91a] [Ghe91b] [CGL92] [Cha92] [CGL93] [Cas93] [CL94].

6.4 Type Reconstruction

Type reconstruction (or type inference) is the process of reconstructing types for a program that has no type declarations. A practical and sound algorithm is described in Milner’s paper “A Theory of Type Polymorphism in Programming” [Mil78]. Another exposition of this material, incorporating refinements found in [DM82], is found in [Car87].

Milner-style type inference system is used in the programming language ML [GMW79] [Mil84] [Har86] [HMM86]. The ideal model has been used to give a semantics to the ML type system [MPS86].
The question of whether one can do type inference for more powerful type systems has been an active area of research (e.g., [McC84] [Wan89]). A recent survey of what is known about such problems is found in [Tiu90].

See Knight’s survey [Kni89] for a discussion of the central role that unification plays in type inference and more details on unification itself.

6.5 Research Languages

The language Russell was developed at Cornell to investigate how types can be treated as values. There are many papers that have appeared about Russell, but perhaps the best introduction to the language is the paper “Data Types are Values” [DD85], which can be consulted for other references. Russell has separate mechanisms for building types and for information hiding; the same idea is found in Haskell [Hud89, Pages 387-388]. Interesting variations are found in Miranda [Tur85].

Much recent work has centered around the language ML and its modern variant Standard ML [Mil84] [MTH90] [MT91]. Besides the type inference in its type system mentioned above, Standard ML has an interesting module system [Mac84] [Har85] [HMT87].

References


