WIKIPEDIA

Denotational semantics

In computer science, **denotational semantics** (initially known as **mathematical semantics** or **Scott–Strachey semantics**) is an approach of formalizing the meanings of programming languages by constructing mathematical objects (called *denotations*) that describe the meanings of expressions from the languages. Other approaches providing formal semantics of programming languages include axiomatic semantics and operational semantics.

Broadly speaking, denotational semantics is concerned with finding mathematical objects called domains that represent what programs do. For example, programs (or program phrases) might be represented by partial functions[1][2] or by games[3] between the environment and the system.

An important tenet of denotational semantics is that *semantics should be compositional*: the denotation of a program phrase should be built out of the denotations of its subphrases.

Contents

Historical development

Denotational semantics originated in the work of Christopher Strachey and Dana Scott published in the early 1970s.[4][5] As originally developed by Strachey and Scott, denotational semantics provided the meaning of a computer program as a function that mapped input into output.[6] To give meanings to recursively defined programs, Scott proposed working with continuous functions between domains, specifically complete partial orders. As described below, work has continued in investigating appropriate denotational semantics for aspects of programming languages such as sequentiality, concurrency, non-determinism and local state.

Denotational semantics has been developed for modern programming languages that use capabilities like concurrency and exceptions, e.g., Concurrent ML,[7] CSP,[8] and Haskell.[9] The semantics of these languages is compositional in that the meaning of a phrase depends on the meanings of its subphrases. For example, the meaning of the applicative expression f(E1,E2) is defined in terms of semantics of its subphrases f, E1 and E2. In a modern programming language, E1 and E2 can be evaluated concurrently and the execution of one of them might affect the other by interacting through shared objects causing their meanings to be defined in terms of each other. Also, E1 or E2 might throw an exception which could terminate the execution of the other one. The sections below describe special cases of the semantics of these modern programming languages.

Meanings of recursive programs

Denotational semantics is ascribed to a program phrase as a function from an environment (holding the current values of its free variables) to its denotation. For example, the phrase `n*m` produces a denotation when provided with an environment that has binding for its two free variables: `n` and `m`. If in the environment `n` has the value 3 and `m` has the value 5, then the denotation is 15.[10]

A function can be represented as a set of ordered pairs of argument and corresponding result values. For example, the set {(0,1), (4,3)} denotes a function with result 1 for argument 0, result 3 for the argument 4, and undefined otherwise.

The problem to be solved is to provide meanings for recursive programs that are defined in terms of themselves such as the definition of the factorial function as

```
int factorial(int n) { if (n == 0) then return 1; else return n * factorial(n-1); }
```

A solution is to build up the meanings by approximation. The factorial function is a total function from ℕ to ℕ (defined everywhere in its domain), but we model it as a partial function. At the beginning, we start with the empty function (an empty set). Next, we add the ordered pair (0,1) to the function to result in another partial function that better approximates the factorial function. Afterwards, we add yet another ordered pair (1,1) to create an even better approximation.

It is instructive to think of this chain of iteration for a "partial factorial function" F as F^0, F^1, F^2, ... where F^n indicates *F* applied *n* times.

- F^0({}) is the totally undefined partial function, represented as the set {};
- F^1({}) is the partial function represented as the set {(0,1)}: it is defined at 0, to be 1, and undefined elsewhere;
- F^5({}) is the partial function represented as the set {(0,1), (1,1), (2,2), (3,6), (4,24)}: it is defined for arguments 0,1,2,3,4.

This iterative process builds a sequence of partial functions from ℕ to ℕ. Partial functions form a chain-complete partial order using ⊆ as the ordering. Furthermore, this iterative process of better approximations of the factorial function forms an expansive (also called progressive) mapping because each $F^i \leq F^{i+1}$ using ⊆ as the ordering. So by a fixed-point theorem (specifically Bourbaki–Witt theorem), there exists a fixed point for this iterative process.

In this case, the fixed point is the least upper bound of this chain, which is the full `factorial` function, which can be expressed as the direct limit

$$\bigsqcup_{i \in \mathbb{N}} F^i(\{\}).$$

Here, the symbol "⊔" is the directed join (of directed sets), meaning "least upper bound". The directed join is essentially the join of directed sets.

Denotational semantics of non-deterministic programs

The concept of power domains has been developed to give a denotational semantics to non-deterministic sequential programs. Writing *P* for a power-domain constructor, the domain *P*(*D*) is the domain of non-deterministic computations of type denoted by *D*.

There are difficulties with fairness and unboundedness in domain-theoretic models of non-determinism.[11]

Denotational semantics of concurrency

Many researchers have argued that the domain-theoretic models given above do not suffice for the more general case of concurrent computation. For this reason various new models have been introduced. In the early 1980s, people began using the style of denotational semantics to give semantics for concurrent languages. Examples include Will Clinger's work with the actor model; Glynn Winskel's work with event structures and petri nets;[12] and the work by Francez, Hoare, Lehmann, and de Roever (1979) on trace semantics for CSP.[13] All these lines of inquiry remain under investigation (see e.g. the various denotational models for CSP[8]).

Recently, Winskel and others have proposed the category of profunctors as a domain theory for concurrency.[14][15]

Denotational semantics of state

State (such as a heap) and simple imperative features can be straightforwardly modeled in the denotational semantics described above. All the textbooks below have the details. The key idea is to consider a command as a partial function on some domain of states. The meaning of "`x:=3`" is then the function that takes a state to the state with `3` assigned to `x`. The sequencing operator "`;`" is denoted by composition of functions. Fixed-point constructions are then used to give a semantics to looping constructs, such as "`while`".

Things become more difficult in modelling programs with local variables. One approach is to no longer work with domains, but instead to interpret types as functors from some category of worlds to a category of domains. Programs are then denoted by natural continuous functions between these functors.[16][17]

Denotations of data types

Many programming languages allow users to define recursive data types. For example, the type of lists of numbers can be specified by

```
datatype list = Cons of nat * list | Empty
```

This section deals only with functional data structures that cannot change. Conventional imperative programming languages would typically allow the elements of such a recursive list to be changed.

For another example: the type of denotations of the untyped lambda calculus is

```
datatype D = D of (D → D)
```

The problem of *solving domain equations* is concerned with finding domains that model these kinds of datatypes. One approach, roughly speaking, is to consider the collection of all domains as a domain itself, and then solve the recursive definition there. The textbooks below give more details.

Polymorphic data types are data types that are defined with a parameter. For example, the type of α `list`s is defined by

```
datatype α list = Cons of α * α list | Empty
```

Lists of natural numbers, then, are of type `nat list`, while lists of strings are of `string list`.

Some researchers have developed domain theoretic models of polymorphism. Other researchers have also modeled parametric polymorphism within constructive set theories. Details are found in the textbooks listed below.

A recent research area has involved denotational semantics for object and class based programming languages.[18]

Denotational semantics for programs of restricted complexity

Following the development of programming languages based on linear logic, denotational semantics have been given to languages for linear usage (see e.g. proof nets, coherence spaces) and also polynomial time complexity.[19]

Denotational semantics of sequentiality

The problem of full abstraction for the sequential programming language PCF was, for a long time, a big open question in denotational semantics. The difficulty with PCF is that it is a very sequential language. For example, there is no way to define the parallel-or function in PCF. It is for this reason that the approach using domains, as introduced above, yields a denotational semantics that is not fully abstract.

This open question was mostly resolved in the 1990s with the development of game semantics and also with techniques involving logical relations.[20] For more details, see the page on PCF.

Denotational semantics as source-to-source translation

It is often useful to translate one programming language into another. For example, a concurrent programming language might be translated into a process calculus; a high-level programming language might be translated into byte-code. (Indeed, conventional denotational semantics can be seen as the interpretation of programming languages into the internal language of the category of domains.)

In this context, notions from denotational semantics, such as full abstraction, help to satisfy security concerns.[21][22]

Abstraction

It is often considered important to connect denotational semantics with operational semantics. This is especially important when the denotational semantics is rather mathematical and abstract, and the operational semantics is more concrete or closer to the computational intuitions. The following properties of a denotational semantics are often of interest.

1. **Syntax independence**: The denotations of programs should not involve the syntax of the source language.
2. **Adequacy (or soundness)**: All observably distinct programs have distinct denotations;
3. **Full abstraction**: All observationally equivalent programs have equal denotations.

For semantics in the traditional style, adequacy and full abstraction may be understood roughly as the requirement that "operational equivalence coincides with denotational equality". For denotational semantics in more intensional models, such as the actor model and process calculi, there are different notions of equivalence within each model, and so the concepts of adequacy and of full abstraction are a matter of debate, and harder to pin down. Also the mathematical structure of operational semantics and denotational semantics can become very close.

Additional desirable properties we may wish to hold between operational and denotational semantics are:

1. **Constructivism**: Constructivism is concerned with whether domain elements can be shown to exist by constructive methods.
2. **Independence of denotational and operational semantics**: The denotational semantics should be formalized using mathematical structures that are independent of the operational semantics of a programming language; However, the underlying concepts can be closely related. See the section on Compositionality below.
3. **Full completeness** or **definability**: Every morphism of the semantic model should be the denotation of a program.[23]

Compositionality

An important aspect of denotational semantics of programming languages is compositionality, by which the denotation of a program is constructed from denotations of its parts. For example, consider the expression "7 + 4". Compositionality in this case is to provide a meaning for "7 + 4" in terms of the meanings of "7", "4" and "+".

A basic denotational semantics in domain theory is compositional because it is given as follows. We start by considering program fragments, i.e. programs with free variables. A *typing context* assigns a type to each free variable. For instance, in the expression (x + y) might be considered in a typing context (x:`nat`,y:`nat`). We now give a denotational semantics to program fragments, using the following scheme.

1. We begin by describing the meaning of the types of our language: the meaning of each type must be a domain. We write $[\![\tau]\!]$ for the domain denoting the type τ. For instance, the meaning of type `nat` should be the domain of natural numbers: $[\![\mathtt{nat}]\!] = \mathbb{N}_\bot$.
2. From the meaning of types we derive a meaning for typing contexts. We set $[\![x_1{:}\tau_1,\ldots, x_n{:}\tau_n]\!] = [\![\tau_1]\!] \times \ldots \times [\![\tau_n]\!]$. For instance, $[\![x{:}\mathtt{nat},y{:}\mathtt{nat}]\!] = \mathbb{N}_\bot\times\mathbb{N}_\bot$. As a special case, the meaning of

the empty typing context, with no variables, is the domain with one element, denoted 1.

3. Finally, we must give a meaning to each program-fragment-in-typing-context. Suppose that P is a program fragment of type σ, in typing context Γ, often written $\Gamma \vdash P{:}\sigma$. Then the meaning of this program-in-typing-context must be a continuous function $[\![\Gamma \vdash P{:}\sigma]\!] : [\![\Gamma]\!] \to [\![\sigma]\!]$. For instance, $[\![\vdash 7{:}\mathtt{nat}]\!]{:}1 \to \mathbb{N}_\perp$ is the constantly "7" function, while $[\![x{:}\mathtt{nat}, y{:}\mathtt{nat} \vdash x{+}y{:}\mathtt{nat}]\!]{:}\mathbb{N}_\perp \times \mathbb{N}_\perp \to \mathbb{N}_\perp$ is the function that adds two numbers.

Now, the meaning of the compound expression (7+4) is determined by composing the three functions $[\![\vdash 7{:}\mathtt{nat}]\!]{:}1 \to \mathbb{N}_\perp$, $[\![\vdash 4{:}\mathtt{nat}]\!]{:}1 \to \mathbb{N}_\perp$, and $[\![x{:}\mathtt{nat}, y{:}\mathtt{nat} \vdash x{+}y{:}\mathtt{nat}]\!]{:}\mathbb{N}_\perp \times \mathbb{N}_\perp \to \mathbb{N}_\perp$.

In fact, this is a general scheme for compositional denotational semantics. There is nothing specific about domains and continuous functions here. One can work with a different category instead. For example, in game semantics, the category of games has games as objects and strategies as morphisms: we can interpret types as games, and programs as strategies. For a simple language without general recursion, we can make do with the category of sets and functions. For a language with side-effects, we can work in the Kleisli category for a monad. For a language with state, we can work in a functor category. Milner has advocated modelling location and interaction by working in a category with interfaces as objects and *bigraphs* as morphisms.[24]

Semantics versus implementation

According to Dana Scott (1980):[25]

> *It is not necessary for the semantics to determine an implementation, but it should provide criteria for showing that an implementation is correct.*

According to Clinger (1981):[26]: 79

> *Usually, however, the formal semantics of a conventional sequential programming language may itself be interpreted to provide an (inefficient) implementation of the language. A formal semantics need not always provide such an implementation, though, and to believe that semantics must provide an implementation leads to confusion about the formal semantics of concurrent languages. Such confusion is painfully evident when the presence of unbounded nondeterminism in a programming language's semantics is said to imply that the programming language cannot be implemented.*

Connections to other areas of computer science

Some work in denotational semantics has interpreted types as domains in the sense of domain theory, which can be seen as a branch of model theory, leading to connections with type theory and category theory. Within computer science, there are connections with abstract interpretation, program verification, and model checking.

References

1. Dana S. Scott. Outline of a mathematical theory of computation (https://ropas.snu.ac.kr/~kwang/520/readings/sco70.pdf). Technical Monograph PRG-2, Oxford University Computing Laboratory, Oxford, England, November 1970.
2. Dana Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages* Oxford Programming Research Group Technical Monograph. PRG-6. 1971.

3. Jan Jürjens. J. Games In The Semantics Of Programming Languages – An Elementary Introduction. Synthese 133, 131–158 (2002). https://doi.org/10.1023/A:1020883810034 (https://doi.org/10.1023/A:1020883810034)
4. Dana S. Scott. Outline of a mathematical theory of computation (https://ropas.snu.ac.kr/~kwang/520/readings/sco70.pdf). Technical Monograph PRG-2, Oxford University Computing Laboratory, Oxford, England, November 1970.
5. Dana Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages* Oxford Programming Research Group Technical Monograph. PRG-6. 1971.
6. Dana Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages* Oxford Programming Research Group Technical Monograph. PRG-6. 1971.
7. John Reppy "Concurrent ML: Design, Application and Semantics" in Springer-Verlag, *Lecture Notes in Computer Science*, Vol. 693. 1993
8. A. W. Roscoe. "The Theory and Practice of Concurrency" Prentice-Hall. Revised 2005.
9. Simon Peyton Jones, Alastair Reid, Fergus Henderson, Tony Hoare, and Simon Marlow. "A semantics for imprecise exceptions (http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.50.1525&rep=rep1&type=pdf)" Conference on Programming Language Design and Implementation. 1999.
10. Dana Scott and Christopher Strachey. *Toward a mathematical semantics for computer languages* Oxford Programming Research Group Technical Monograph. PRG-6. 1971.
11. Levy, Paul Blain (2007). "Amb Breaks Well-Pointedness, Ground Amb Doesn't" (https://doi.org/10.1016%2Fj.entcs.2007.02.036). *Electr. Notes Theor. Comput. Sci*. **173**: 221–239. doi:10.1016/j.entcs.2007.02.036 (https://doi.org/10.1016%2Fj.entcs.2007.02.036).
12. *Event Structure Semantics for CCS and Related Languages (https://www.cl.cam.ac.uk/~gw104/eventStructures82.pdf)*. DAIMI Research Report, University of Aarhus, 67 pp., April 1983.
13. Nissim Francez, C. A. R. Hoare, Daniel Lehmann, and Willem-Paul de Roever. "Semantics of nondeterminism, concurrency, and communication (https://dspace.library.uu.nl/bitstream/handle/1874/24888/francez_79_Semantics+of+nondeterminism.pdf?sequence=1)", *Journal of Computer and System Sciences*. December 1979.
14. Cattani, Gian Luca; Winskel, Glynn (2005). "Profunctors, open maps and bisimulation". *Mathematical Structures in Computer Science*. **15** (3): 553–614. CiteSeerX 10.1.1.111.6243 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.111.6243). doi:10.1017/S0960129505004718 (https://doi.org/10.1017%2FS0960129505004718).
15. Nygaard, Mikkel; Winskel, Glynn (2004). "Domain theory for concurrency" (https://doi.org/10.1016%2Fj.tcs.2004.01.029). *Theor. Comput. Sci*. **316** (1–3): 153–190. doi:10.1016/j.tcs.2004.01.029 (https://doi.org/10.1016%2Fj.tcs.2004.01.029).
16. Peter W. O'Hearn, John Power, Robert D. Tennent, Makoto Takeyama. Syntactic control of interference revisited. *Electr. Notes Theor. Comput. Sci.* 1. 1995.
17. Frank J. Oles. *A Category-Theoretic Approach to the Semantics of Programming*. PhD thesis, Syracuse University, New York, USA. 1982.
18. Reus, Bernhard; Streicher, Thomas (2004). "Semantics and logic of object calculi" (https://doi.org/10.1016%2Fj.tcs.2004.01.030). *Theor. Comput. Sci*. **316** (1): 191–213. doi:10.1016/j.tcs.2004.01.030 (https://doi.org/10.1016%2Fj.tcs.2004.01.030).
19. Baillot, P. (2004). "Stratified coherence spaces: a denotational semantics for Light Linear Logic" (https://doi.org/10.1016%2Fj.tcs.2003.10.015). *Theor. Comput. Sci*. **318** (1–2): 29–55. doi:10.1016/j.tcs.2003.10.015 (https://doi.org/10.1016%2Fj.tcs.2003.10.015).
20. O'Hearn, P.W.; Riecke, J.G. (July 1995). "Kripke Logical Relations and PCF" (https://surface.syr.edu/lcsmith_other/3). *Information and Computation*. **120** (1): 107–116. doi:10.1006/inco.1995.1103 (https://doi.org/10.1006%2Finco.1995.1103).
21. Martin Abadi. "Protection in programming-language translations". *Proc. of ICALP'98*. LNCS 1443. 1998.

22. Kennedy, Andrew (2006). "Securing the .NET programmingmodel" (https://doi.org/10.1016%2Fj.tcs.2006.08.014). *Theor. Comput. Sci*. **364** (3): 311–7. doi:10.1016/j.tcs.2006.08.014 (https://doi.org/10.1016%2Fj.tcs.2006.08.014).
23. Curien, Pierre-Louis (2007). "Definability and Full Abstraction" (https://doi.org/10.1016%2Fj.entcs.2007.02.011). *Electronic Notes in Theoretical Computer Science*. **172**: 301–310. doi:10.1016/j.entcs.2007.02.011 (https://doi.org/10.1016%2Fj.entcs.2007.02.011).
24. Milner, Robin (2009). *The Space and Motion of Communicating Agents*. Cambridge University Press. ISBN 978-0-521-73833-0. 2009 draft (https://blog.itu.dk/SMDS-F2010/files/2010/04/milner-2009-the-space-and-motion-of-communicating-agents.pdf) Archived (https://web.archive.org/web/20120402095417/https://blog.itu.dk/SMDS-F2010/files/2010/04/milner-2009-the-space-and-motion-of-communicating-agents.pdf) 2012-04-02 at the Wayback Machine.
25. "What is Denotational Semantics?", MIT Laboratory for Computer Science Distinguished Lecture Series, 17 April 1980, cited in Clinger (1981).
26. Clinger, William D. (1981). "Foundations of Actor Semantics" (PhD). Massachusetts Institute of Technology. hdl:1721.1/6935 (https://hdl.handle.net/1721.1%2F6935). AITR-633.

Further reading

Textbooks

- Milne, R.E.; Strachey, C. (1976). *A theory of programming language semantics*. ISBN 978-1-5041-2833-9.
- Gordon, M.J.C. (2012) [1979]. *The Denotational Description of Programming Languages: An Introduction* (https://books.google.com/books?id=s4jTBwAAQBAJ). Springer. ISBN 978-1-4612-6228-2.
- Stoy, Joseph E. (1977). *Denotational Semantics: The Scott-Strachey Approach to Programming Language Semantics*. MIT Press. ISBN 978-0262191470. (A classic if dated textbook.)
- Schmidt, David A. (1986). *Denotational semantics: a methodology for language development*. Allyn & Bacon. ISBN 978-0-205-10450-5. (out of print now; free electronic version available (http://www.cis.ksu.edu/~schmidt/text/densem.html))
- Gunter, Carl (1992). *Semantics of Programming Languages: Structures and Techniques*. MIT Press. ISBN 978-0262071437.
- Winskel, Glynn (1993). *Formal Semantics of Programming Languages*. MIT Press. ISBN 978-0262731034.
- Tennent, R.D. (1994). "Denotational semantics". In Abramsky, S.; Gabbay, Dov M.; Maibaum, T.S.E. (eds.). *Semantic Structures*. Handbook of logic in computer science. **3**. Oxford University Press. pp. 169–322. ISBN 978-0-19-853762-5.
- Abramsky, S.; Jung, A. (1994). "Domain theory" (http://www.cs.bham.ac.uk/~axj/pub/papers/handy1.pdf) (PDF). *Abramsky, Gabbay & Maibaum 1994*.
- Stoltenberg-Hansen, V.; Lindström, I.; Griffor, E.R. (1994). *Mathematical Theory of Domains* (https://archive.org/details/mathematicaltheo0000stol). Cambridge University Press. ISBN 978-0-521-38344-8.

Lecture notes

- Winskel, Glynn. "Denotational Semantics" (http://www.cl.cam.ac.uk/~gw104/dens.pdf) (PDF). University of Cambridge.

Other references

- Greif, Irene (August 1975). *Semantics of Communicating Parallel Processes* (https://dspace.mit.edu/bitstream/handle/1721.1/57710/02200859-MIT.pdf) (PDF) (PhD). Project MAC. Massachusetts Institute of Technology. ADA016302.
- Plotkin, G.D. (1976). "A powerdomain construction". *SIAM J. Comput*. **5** (3): 452–487. CiteSeerX 10.1.1.158.4318 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.158.4318). doi:10.1137/0205035 (https://doi.org/10.1137%2F0205035).
- Edsger Dijkstra. *A Discipline of Programming* Prentice Hall. 1976.
- Krzysztof R. Apt, J. W. de Bakker. *Exercises in Denotational Semantics* MFCS 1976: 1-11
- de Bakker, J.W. (1976). "Least Fixed Points Revisited" (https://doi.org/10.1016%2F0304-3975%2876%2990031-1). *Theor. Comput. Sci*. **2** (2): 155–181. doi:10.1016/0304-3975(76)90031-1 (https://doi.org/10.1016%2F0304-3975%2876%2990031-1).
- Smyth, Michael B. (1978). "Power domains" (https://doi.org/10.1016%2F0022-0000%2878%2990048-X). *J. Comput. Syst. Sci*. **16**: 23–36. doi:10.1016/0022-0000(78)90048-X (https://doi.org/10.1016%2F0022-0000%2878%2990048-X).
- Francez, Nissim; Hoare, C.A.R.; Lehmann, Daniel; de Roever, Willem-Paul (December 1979). *Semantics of nondeterminism, concurrency, and communication*. *Journal of Computer and System Sciences*. Lecture Notes in Computer Science. **64**. pp. 191–200. doi:10.1007/3-540-08921-7_67 (https://doi.org/10.1007%2F3-540-08921-7_67). hdl:1874/15886 (https://hdl.handle.net/1874%2F15886). ISBN 978-3-540-08921-6.
- Lynch, Nancy; Fischer, Michael J. (1979). "On describing the behavior of distributed systems" (https://books.google.com/books?id=lLgqAQAAIAAJ). In Kahn, G. (ed.). *Semantics of concurrent computation: proceedings of the international symposium, Évian, France, July 2-4, 1979*. Springer. ISBN 978-3-540-09511-8.
- Schwartz, Jerald (1979). "Denotational semantics of parallelism". *Kahn 1979*.
- Wadge, William (1979). "An extensional treatment of dataflow deadlock". *Kahn 1979*.
- Ralph-Johan Back. "Semantics of Unbounded Nondeterminism" ICALP 1980.
- David Park. *On the semantics of fair parallelism (https://link.springer.com/content/pdf/10.1007/3-540-10007-5_47.pdf) Proceedings of the Winter School on Formal Software Specification*. Springer-Verlag. 1980.
- Clinger, W.D. (1981). "Foundations of Actor Semantics" (PhD). Massachusetts Institute of Technology. hdl:1721.1/6935 (https://hdl.handle.net/1721.1%2F6935). AITR-633.
- Allison, L. (1986). *A Practical Introduction to Denotational Semantics* (https://books.google.com/books?id=uIdF11msK58C). Cambridge University Press. ISBN 978-0-521-31423-7.
- America, P.; de Bakker, J.; Kok, J.N.; Rutten, J. (1989). "Denotational semantics of a parallel object-oriented language" (https://ir.cwi.nl/pub/1602). *Information and Computation*. **83** (2): 152–205. doi:10.1016/0890-5401(89)90057-6 (https://doi.org/10.1016%2F0890-5401%2889%2990057-6).
- Schmidt, David A. (1994). *The Structure of Typed Programming Languages*. MIT Press. ISBN 978-0-262-69171-0.

External links

- *Denotational Semantics* (http://www.csse.monash.edu.au/~lloyd/tilde/Semantics/). Overview of book by Lloyd Allison
- Schreiner, Wolfgang (1995). "Structure of Programming Languages I: Denotational Semantics" (http://www.risc.uni-linz.ac.at/people/schreine/courses/densem/densem.html). *Course notes*.