A Technique for Defining Predicate Transformers

Wolfgang Polak

TR 17-4
October 1988

Odyssey Research Associates
301A Harris B. Dates Drive
Ithaca, New York 14850-1313
Tel. (607)–277–2020

A Technique for Defining Predicate Transformers*

Extended Abstract

Wolfgang Polak

October 21, 1988

Abstract

Denotational semantics is a general technique for formally defining programming languages and for reasoning about their properties. But a logical approach is often more convenient for reasoning about individual programs. This paper presents a framework for systematically deriving predicate transformers for procedural languages from a continuation semantics. Many idioms of denotational semantics, such as environments and various forms of continuations, are employed in the resulting predicate transformer definition. Thus, language features such as side effects and exception handling become amenable to a logical definition. The soundness of derived predicate transformers follows from the correctness of local transformation. Elements of such a soundness proof are demonstrated.

*This research was prepared for Odyssey Research Associates and has been sponsored by the USAF, Rome Air Development Center under contract number F30602-86-C-0071

1 Introduction

Denotational semantics is a very general technique for formally defining programming languages and for reasoning about their properties. Simpler logical formalisms are often sufficient and more convenient for reasoning about particular programs of a language. Verification systems, for example, are often based on a logical definition of the underlying language. In a logical or assertion-based formalism the effect of a program is described by predicates on the computation state at various points in the program (e.g. [5]). The effect of a language fragment can be characterized by pre– and postconditions. The relation between pre– and postconditions can be defined axiomatically (e.g. [9]) or through a function mapping one to the other (e.g. [4]). In this paper predicate transformers are functions mapping postconditions to sufficient preconditions. Preconditions in our formalism are "weak", i.e. they do not ensure termination of the program.

The definition of proof rules and predicate transformers for new language constructs is often done in an ad hoc manner, or with unrealistic assumptions about the language being defined. In contrast, researchers in denotational semantics have developed a large number of idioms and general mechanisms that make the definition of even complex languages (tedious) routine. The results presented here allow the systematic derivation of predicate transformers from a continuation semantics, thus making use of such definitional idioms. For instance, the use of expression continuations makes possible the description of side effects, errors during expression evaluation, and exception handling mechanisms.

The concept of a *derived definition* is introduced to formally describe the relation between a denotational semantics and a structurally similar definition. For instance, a stack semantics (see e.g. [11]) may be viewed as *derived* from a continuation semantics.

A predicate on computation states can be viewed as a precondition on the remainder of the computation. In this sense a predicate describes a continuation. This relation between continuations and predicates is formally defined, generalizing results of [14]. It is shown that predicate transformers can be defined as a derived definition of a continuation semantics.

Details of such a predicate transformer definition are illustrated with a small but nontrivial example language. The use of the logical equivalents of environments, expression continuations, and procedure values is demonstrated with the example.

By the nature of their construction, the soundness of the predicate transformers follows from a simple induction argument. Key steps of such a proof are presented for the example language.

The technique has been successfully applied to define and implement predicate transformers for a large subset of sequential Ada ([15]).

2 Predicate Transformers

2.1 Assertion Language

An *assertion language* is a logical language $\mathcal{A}$ for describing the semantics of the data objects of a programming language. Pre- and postconditions as well as verification conditions are expressed in $\mathcal{A}$. Predicate transformers are syntactic transformation of terms of $\mathcal{A}$.

An *annotation* is a syntactic extension of the programming language that provides a means to associate terms of $\mathcal{A}$ with a specific language construct. Annotations are used to provide induction hypotheses for iteration and recursion constructs. Such annotations affect the precondition generated for a program but not its semantics. Regardless of the annotations, only sound preconditions are generated. Insufficient annotations result in preconditions that are too strong (e.g. *false*).

The semantic definition of any programming language needs to deal with run-time errors arising as the result of illegal operations. Some languages provide error recovery mechanisms that allow the program to continue (e.g. exception handling). In this case, a run-time error may cause a change in control flow. To define the semantics of errors other researchers (e.g. [1,3,13]) have used assertion languages that include error conditions, exception values, or undefined values. The resulting logical system embody parts of the programming language semantics and/or will complicate the reasoning about the assertion language. Yet, even with such extensions the description of control flow changes due to errors is difficult.

For the kind of predicate transformers described here a two-valued logic suffices. The assertion language must be rich enough to describe (i) the values returned by programming language expressions in the case of error-free execution and (ii) the preconditions under which evaluation of an expression results in an error. Predicate transformers for expression evaluation will generate either preconditions to ensure error-free execution, or conditional terms describing both normal execution and error recovery (see section 3.2).

In section 3 we assume $\mathcal{A}$ to be a logical language with an unbounded set of variables $x \in X$. $\mathcal{T}(X)$ is the set of terms (over X), $\mathcal{T}$ is the set of ground terms, and formulas are boolean terms $\mathcal{T}_B(X)$. In addition to arithmetic and boolean operators $\mathcal{A}$ contains conditionals *if* _ *then* _ *else* _. Terms for array selection _ [_] and array update (_; _ : _) are defined as in [6]. Arrays in the assertion language are unbounded and can be indexed by any integer. Preconditions generated for the evaluation of the programming language expressions `a[i]` will guarantee that the index is within the appropriate range.

If $op(\_,\_)$ is an operator symbol in $\mathcal{A}$ then $\boxed{op} : \mathcal{T}(X), \mathcal{T}(X) \to \mathcal{T}(X)$ is a constructor function; i.e. given two terms t_1 and t_2, $\boxed{op}\, t_1 t_2$ is a term that is the application of op to t_1 and t_2. Function $subst \in \mathcal{T}(X) \to X \to \mathcal{T}(X) \to \mathcal{T}(X)$ defines substitution on terms, i.e. $subst\ p\ x\ v$ denotes the substitution of v for free occurrences of x in p. Let $\vec{x}$ be a set of variables, the function $\boxed{\forall}\, \vec{x}.t$ constructs a universally quantified term. Terms of $\mathcal{A}$ are

interpreted over some fixed structure; $\models t$ is the value of t in this structure.

Results presented here will extend to richer languages. For instance, the assertion language may contain mechanisms to define new theories for abstract data types (e.g. *Clear* [2], *Larch* [8]).

2.2 Denotational Semantics

Familiarity with the basic concepts of continuation semantics (e.g. [18]) is assumed. This section summarizes the notation used and introduces the concept of derived definitions.

Domains are complete partial orders ([18]). For a set S, $S_\perp$ is the flat domain. Operators $\rightarrow$, $+$, and $\times$ construct function, sum, and product domains respectively. Elements of abstract syntax domains are inclosed in brackets, as in $[\![\mathbf{x + y}]\!]$.

Functions are defined in typed lambda calculus. In most cases type information is clear from the context and will be omitted. Functions are curried, and application is left associative e.g. $fgx = (fg)x$. To avoid excessive parentheses the ";" operator indicates application but has lower precedence than juxtaposition and associates to the right, e.g. $f;gx;hy = f(gx(hy))$. Lambda abstraction is of lowest precedence, i.e. $\lambda x.f;g;h$ means $\lambda x.(f;g;h)$.

If $f \in A \rightarrow B$, $a \in A$, and $b \in B$ then $f[a/b]$ is a function such that $f[a/b]a = b$ and $f[a/b]a' = fa'$ if $a \neq a'$. For a tuple $p \in A \times B$ the notation p^i selects the i-th component of p. For a sum $p \in A + B$ then $p|A$ is the projection of p to A.

A denotational semantic definition can itself be viewed as a syntactic object consisting of a set of domains D_i, a number of constant symbols $d_j \in D_i$ and a set of terms Δ_k of typed lambda calculus denoting the meaning functions. We are interested in mapping a denotational semantics into a *derived* definition, i.e. a structurally similar definition that uses different domains. In particular, we shall see that predicate transformers can be defined as a derived definition from a continuation semantics.

Let P_i be a set of (different) domains. Further, let $\rightsquigarrow$ be a mapping that assigns to each domain D_i a domain P_i, and assigns to each constant $d_j \in D_i$ a constant $p_j \in P_i$. We write $D_i \rightsquigarrow P_i$ and $d_j \rightsquigarrow p_j$ respectively. A mapping $\rightsquigarrow$ extends to a transformation on terms of a denotational definition in the obvious way, we write $\Delta \rightsquigarrow \Pi$.

Let $\mathcal{R}_{D_i} \subset D_i \times P_i$ be relations[1] such that

$$\mathcal{R}_{D_1 \rightarrow D_2}(d,p) ::= \forall d_1, p_1.\mathcal{R}_{D_1}(d_1,p_1) \Rightarrow \mathcal{R}_{D_2}(d\ d_1, p\ p_1) \quad (1)$$

$$\mathcal{R}_{D_1 + D_2}(d,p) ::= \mathcal{R}_{D_1}(d|D_1, p|P_1) \vee \mathcal{R}_{D_2}(d|D_2, p|P_2) \quad (2)$$

$$\mathcal{R}_{D_1 \times D_2}(d,p) ::= \mathcal{R}_{D_1}(d^1,p^1) \wedge \mathcal{R}_{D_2}(d^2,p^2) \quad (3)$$

For $D = D_1 \rightarrow D_2$, with $D \rightsquigarrow P$, the domain P need not necessarily be a function domain. If it is not, there must be a function that serves the role of application, e.g. $apply_P \in$

[1] Care must be taken to ensure that recursive definitions of such relations are well defined (see e.g. [11]).

$P \to P_1 \to P_2$. Terms of the form $d\,d_1$ are mapped to *apply* $p\ p_1$ where $d \rightsquigarrow p$ and $d_1 \rightsquigarrow p_1$. Similarly, terms in D that are constructed with lambda abstraction need to be mapped into suitable combinators in P (see 2.3, 3.3).

Lemma 1: If for all constants $d \rightsquigarrow p$ relation[2] $\mathcal{R}(d,p)$ is true then for any pair of closed terms $\Delta \rightsquigarrow \Pi$ relation $\mathcal{R}(\Delta,\Pi)$ holds.

The lemma can be proved by structural induction (see full paper for details).

2.3 Predicates as Continuations

The key idea (following [14]) is to define a transformation $\rightsquigarrow$ that maps the domain of continuations to a domain of predicates. As a result, the meaning functions of the continuation semantics (which are continuation transformers) become predicate transformers. This section describes the domains of a predicate transformer semantics and their relation $\mathcal{R}$ to continuation semantics. Where necessary subscripts $\mathcal{D}$ and $\mathcal{P}$ are used to distinguish domains from the denotational semantics ($\mathcal{D}$) and predicate transformer definition ($\mathcal{P}$).

Locations are mapped to the domains of variables $X_\perp$ of the assertion language ($L \rightsquigarrow X_\perp$). Intuitively, variables serve as names for locations. The relation $\mathcal{R}_L \subset L \times X_\perp$ is implicitly defined by the respective *new* functions in the two definitions, i.e., if $new \in S \to L$, then $\mathcal{R}_L$ is defined such that $\mathcal{R}_{S\to L}(new_\mathcal{D}, new_\mathcal{P})$.

Values are mapped to ground terms of $\mathcal{A}$, i.e. $V \rightsquigarrow \mathcal{T}_\perp$. $\mathcal{R}_V(\nu,v)$ holds for $\nu \in V$ and term v if either $\nu = \perp$ or ν is the value denoted by v, i.e. $\nu = (\models v)$. To describe abstract data types one may define $h\nu = (\models v)$, where h is an abstraction function.

States are defined as mappings from locations to values ($S_\mathcal{D} = L \to V$). We take $S_\mathcal{P} = X_\perp \to T_\perp$, i.e. a state in the predicate transformer definition assigns a ground term to each variable. $\mathcal{R}_S$ is defined as required by (1).

Answers $A_\mathcal{D}$ are often not further specified. We let $A \rightsquigarrow \{T,F\}_\perp$ and define $\mathcal{R}_A(\alpha,a)$ relative to a subset $A_0 \subset A$ as $(a = T) \Rightarrow (\alpha \in A_0)$. The set A_0 may be arbitrary but must contain $\perp$ and must be closed under limits. Intuitively, A_0 represents a set of "acceptable" answers. An answer in the predicate transformation is true if the answer in the continuation semantics is acceptable.

Continuations are of the form $C = S \to A$, we let $C \rightsquigarrow P = T_B(X)_\perp$, i.e. they are mapped to formulas of the assertion language. Relation $\mathcal{R}_C \subset C \times P$ is defined as

$$\mathcal{R}_C(\theta,p) ::= \forall\sigma,s.\mathcal{R}_S(\sigma,s) \Rightarrow \mathcal{R}_A(\theta\sigma, s \models p)$$

[2]Subscripts of $\mathcal{R}$ are omitted whenever the corresponding domain is clear from the context.

where $s \models p$ is the truth of p when every free variable x in p is replaced by the closed term sx. Note, that P is not a function domain. Instead, application in P is taken to be $\models$. Special rules define transformation of terms of the form $\lambda\sigma.T(\sigma)$ (see section 3.3).

Function Values will be explained in the full paper.

Other Domains For all other domains $\mathcal{R}$ is defined as required by (1), (2), and (3) above.

In continuation semantics statements (Stm) are defined by $D_{Stm} \in Stm \to E_{\mathcal{D}} \to C \to C$. The transformed definition is $M_{Stm} \in Stm \to E_{\mathcal{P}} \to P \to P$. Here $E_{\mathcal{D}}$ and $E_{\mathcal{P}}$ are environments defining context dependent bindings. For fixed statement S and environment $e \in E_{\mathcal{P}}$ the term $M_{Stm}[\![\mathtt{S}]\!]e$ is a predicate transformer $\psi \in P \to P$. The full paper shows that ψ is a predicate transformer in the usual sense: if ψp holds before execution of S and if S terminates then p holds for the state after S.

A predicate transformer definition is sound with respect to a denotational definition if for arbitrary A_0 as above, and for any statement $\mathtt{S} \in Stm$ it can be shown that

$$\mathcal{R}_{E\to C\to C}(D_{Stm}[\![\mathtt{S}]\!], M_{Stm}[\![\mathtt{S}]\!]).$$

By lemma (1) it is only necessary to show that the mapping of all constants preserve the relation $\mathcal{R}$.

3 Example Definition

This section shows an example of a predicate transformer definition that is derived from a continuation semantics. Definitional clauses of the continuation semantics are only shown where they differ from those in the predicate transformers. Unless otherwise noted, all symbols refer to the predicate transformer semantics. The full paper contains a more realistic example language including recursive functions with side effects.

3.1 Statements

The functionality of the predicate transformer for statements (Stm) is $M_{Stm} \in Stm \to E \to P \to P$, i.e. given a statement S, an environment e, and a postcondition p, $M_{Stm}[\![\mathtt{S}]\!]ep$ is a precondition of S. For example, predicate transformers for composition and the empty statement are defined as

$$M_{Stm}[\![\mathtt{S1;\ S2}]\!]ep = M_{Stm}[\![\mathtt{S1}]\!]e; M_{Stm}[\![\mathtt{S2}]\!]ep$$
$$M_{Stm}[\![\mathtt{null}]\!]ep = p$$

Environments define context dependent bindings. For instance, a loop name is bound to the loop postcondition, i.e. $e \in E = (I \rightarrow P)$ where $I \in Id$ is the domain of program identifiers. The `exit` statement can be defined as

$$M_{Stm}[\![\texttt{exit I}]\!]ep = e[\![\texttt{I}]\!]$$

The proper binding of `I` will be established by the definition of the loop statement.

Similarly, `gotos` can be described by binding labels to the precondition of their definitions; `exceptions` use a binding of exception identifiers to the precondition of the appropriate handlers.

3.2 Expressions

Expression continuations are of the form $K_{\mathcal{D}} = V_{\mathcal{D}} \rightarrow C_{\mathcal{D}}$. The corresponding *Expression predicates* are $(K = V \rightarrow P)$. It might help the intuition to think of an expression predicate as a term of the form "some x such that $p(x)$". Predicate transformers for expressions (Exp) are defined by $M_{Exp} \in Exp \rightarrow E \rightarrow K \rightarrow P$.

In addition to handling arbitrary side effects in expression evaluation, expression predicates allow the insertion of error checking clauses in preconditions for all subexpressions. This is illustrated with the definition of array subscripting.

$$M_{Exp}[\![\texttt{N [E]}]\!]ek = M_{Exp}[\![\texttt{N}]\!]e; \lambda u.M_{Exp}[\![\texttt{E}]\!]e; check\ (type\ [\![\texttt{N}]\!]); \lambda v.k(\boxed{\_[\_]}uv)$$

Function *check* inserts the condition that the given index is within the bounds of the array. The definition

$$check\ t = \lambda k.\lambda v.(inrange\ tv)\boxed{\wedge}(kv)$$

will generate preconditions that ensure that all index operations are legal. Function *inrange tv* returns a term of the assertion language that expresses this condition. Alternatively, the definition

$$check\ t = \lambda k.\lambda v.\boxed{if}(inrange\ tv)\boxed{then}(kv)\boxed{else}error$$

describes a semantics whereby computation continues with continuation *error* after an illegal indexing operation is encountered.

3.3 Declarations

The full paper defines the logical equivalent of declaration continuations.

3.4 Locations

Locations are taken to be variables ($x \in X$) of the assertion language. The environment maps program variables to their location (i.e., associated logical variable). In particular, different program variables may map to the same logical variable. Thus, the description of aliasing is possible in principle[3].

Values are ground terms, i.e. $v \in V = \mathcal{T}(X)_\perp$. Structured data objects are stored in single locations. For instance, an assignment to a component of an array is described as assignment to the whole array, e.g. using terms of the form $(\_;\_:\_)$. Similar constructors for updates of other structured objects are well known ([6,10]).

In continuation semantics assignments are defined by a function $update_{\mathcal{D}} \in C \to L \to K$ as $update_{\mathcal{D}}\theta\epsilon = \lambda\nu.\lambda\sigma.\theta(\sigma[\epsilon/\nu])$. The logical correspondent is $update_{\mathcal{P}} \in P \to L \to K$ with $update_{\mathcal{P}}p\ x = \lambda v.subst\ p\,x\,v$, i.e. the precondition of p before assigning a new value, say term v, to a location x is given by the substitution of v for x in p. Soundness of this transformation requires a proof of $\mathcal{R}_{C\to L\to K}(update_{\mathcal{D}}, update_{\mathcal{P}})$. The full paper will contain a proof of this lemma, a more complete definition of the semantics of assignment, and the treatment of data structures.

3.5 Iteration and Recursion

In denotational semantics, iteration and recursion are defined by least fixed points. Because of the strictness of constructors for terms in $\mathcal{A}$, least fixed points are trivial in P. Instead, the operator *fix* is transformed to a different function *induct* that defines a precondition by induction.

Let $fix \in (C \to C) \to C$, we define $fix \rightsquigarrow induct$, where $induct \in (P \to P) \to P$ is given by

$$induct\ \mathcal{G} ::= p_i \boxed{\wedge} (\boxed{\forall} \vec{x}.p_i \boxed{\Rightarrow} (\mathcal{G}p_i))$$

Here, p_i is a (proposed) loop invariant. The origin of p_i is not subject of this paper, it may be given as a program annotation, or may be constructed with heuristics. But note, that *induct* is sound for arbitrary p_i. The quantification ranges over all variables $\vec{x}$ that are subject to substitution in $\mathcal{G}$.

The conjunct $\boxed{\forall} \vec{x}.p_i \boxed{\Rightarrow} (\mathcal{G}p_i)$ corresponds to a verification condition for the induction step of a proof of the invariance of p_i. One might consider proving the universal closure of $p_i \boxed{\Rightarrow} \mathcal{G}p_i$ and simplifying the precondition generated by *induct*. In practice, the more restricted quantification results in a weaker precondition and simplifies loop invariants since trivially invariant properties need not be mentioned.

[3]The practical problem in describing aliasing is the logical definition of suitable abstractions for procedures.

Using *induct*, the semantics of loops becomes

$$M_{Stm}[\![\texttt{I: loop S end loop I}]\!]ep = induct\,;\lambda p'.M_{Stm}[\![\texttt{S}]\!](e[[\![\texttt{I}]\!]/p])p'$$

Soundness of this transformation of *fix* requires proof of the following lemma.

Fixed Point Lemma: Given $\mathcal{F} \in C \to C$ and $\mathcal{G} \in P \to P$ such that $\mathcal{R}_{(C\to C)}(\mathcal{F},\mathcal{G})$, then

$$\mathcal{R}_C(fix\ \mathcal{F}, induct\ \mathcal{G}).$$

The proof will be contained in the full paper. Further, we shall see how a similar transformation can be defined for fixed points of environments that arise in the definition of jumps and recursive procedures.

4 Conclusions

Idioms from denotational semantics can be used in defining predicate transformers for procedural languages. Such a predicate transformer definition can be derived systematically from a continuation semantics. The soundness of the derived definition follows from a set of lemmas showing the soundness of local transformations. The method is applicable with a wide variety of assertion languages. In particular, conventional 2-valued logic may be used, and mechanisms for abstract data type definition can be supported.

The technique described in this paper has been applied successfully to define predicate transformers for most of the sequential part of Ada. David Guaspari provided the logical definition of Ada's data types in the assertion language ([7]). Most of Ada's constraint checking and exception handling mechanism could be described easily[4]. While no formal soundness proof has been given, the concise notation resulted in a relatively short and easy to read definition ([15]). Carla Marceau and Norman Ramsey have implemented a verification condition generator based on this predicate transformer definition. The implementation uses an attribute grammar where predicates and environments are attributes of abstract syntax trees. The Cornell Synthesizer Generator ([16]) provided a convenient framework for the implementation.

Other authors have used syntactic transformations of denotational language definitions to derive related artifacts. Examples are type checking algorithms (e.g. [12]), and code generators (e.g. [19]). It may be worthwhile to study and formalize the relationship between related definitions. For instance, Robinson ([17]) studies the relation between denotational and axiomatic semantics in a category theory setting.

[4]Notable omissions are storage errors and numeric errors.

Acknowledgements

This paper has been greatly improved by numerous suggestions from David Guaspari.

References

[1] H. Barringer, J.H. Cheng, C.B. Jones, A Logic Covering Undefinedness in Program Proofs, *Acta Informatica*, 21, pp 251–269, Springer Verlag 1984.

[2] Rod Burstall and Joseph Goguen, The Semantics of Clear, a Specification Language, in *Proceedings of the 1979 Copenhagen Winter School on Abstract Software Specification*, Dines Bjørner, Ed., Lecture Notes in Computer Science, Vol. 86, pp 292–332, Springer Verlag, 1980

[3] Ole-Johan Dahl, Dag F. Langmyhr, Olaf Owe, *Preliminary Report on the Specification and Programming Language ABEL*, Research Report No. 106, Institute for Informatics, University of Olso, December 1986

[4] Dijkstra, E. W., Guarded commands, nondeterminacy and the formal derivation of programs, *Comm. of the ACM*, 18(8), pp 453–457, August 1975.

[5] Floyd, R. W., Assigning Meaning to Programs, *Proc. Symp. in Applied Mathematics 19, pp 19–32* (J.T. Schwartz, ed.) American Mathematical Society, Providence, R.I., 1969

[6] David Gries, *The Science of Programming*, Springer Verlag, 1981

[7] D. Guaspari, *Domains for Ada Types*, Internal Report, Odyssey Research Assoc., in preparation.

[8] Guttag, J. ,V., Horning, J. J., Wing, J. M., *Larch in five easy pieces*, DEC Systems Research Center, July 1985

[9] C. A. R. Hoare, An Axiomatic Basis of Computer Programming, *Comm. of the ACM*, 12(10), pp 576–580

[10] D. C. Luckham, et.al., *Stanford Pascal Verifier User Manual*, Report No. STAN–CS–79–731, Stanford University, March 1979

[11] Robert Milne, Christopher Strachey, *A theory of programming language semantics*, Chapman and Hall, 1976

[12] Margaret Montenyohl, Mitchell Wand, Correct Flow Analysis in Continuation Semantics, *Fifteenth Annual ACM Symposium on Principles of Programming Languages*, San Diego, January 1988, pp 204–218

[13] Olaf Owe, *An Approach to Program Reasoning Based on a First Order Logic for Partial Functions*, Technical Report CS-081, University of California, San Diego, June 1984, revised February 1985

[14] Wolfgang Polak, Program Verification Based on Denotational Semantics, *Eighth annual ACM Symposium on Principles of Programming Languages*, Williamsburg, January 1981, pp 149–158

[15] Wolfgang Polak, *Predicate Transformer Semantics for Ada*, Internal Report, Odyssey Research Associates, in preparation.

[16] Reps, T. and Teitelbaum, T., The Synthesizer Generator, *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, April 1984, in SIGPLAN Notices 19(5), May 1984

[17] E. Robinson, Logical Aspects of Denotational Semantics, *Category Theory and Computer Science*, Edinburgh, September 1987, pp 238–253, in Lecture Notes in Computer Science, Vol 283, Springer Verlag

[18] Joseph E. Stoy, *Denotational Semantics: The Scott–Strachey Approach to Programming Language Theory*, MIT Press, 1977

[19] Mitchell Wand, Deriving Target Code as a Representation of Continuation Semantics, *ACM Transactions on Programming Languages and Systems*, 4(3), pp 496–517, July 1982.