, , 1–8 ()

Definitional Interpreters Revisited

JOHN C. REYNOLDS john.reynolds@cs.cmu.edu
School of Computer Science, Carnegie Mellon University

Abstract. To introduce the republication of "Definitional Interpreters for Higher-Order Programming Languages", the author recounts the circumstances of its creation, clarifies several obscurities, corrects a few mistakes, and briefly summarizes some more recent developments.

Keywords: operational semantics, denotational semantics, interpreter, lambda calculus, applicative language, functional language, metacircularity, higher-order function, defunctionalization, closure, call by value, call by name, continuation, continuation-passing-style transformation, LISP, ISWIM, PAL, Scheme, SECD machine, J-operator, escape, assignment.

In late 1971, Jean Sammet and Burt Leavenworth asked Art Evans and me to give a tutorial session on "the application of the lambda calculus to programming languages" at the 25th Anniversary ACM National Conference to be held the following summer. I had recently returned from a sabbatical at Queen Mary College in London, where I had immersed myself in Dana Scott's "lattice-theoretic" approach to denotational semantics, Peter Landin's SECD-machine approach to operational semantics, and the ideas about continuations of Lockwood Morris, Christopher Wadsworth, and Jim Morris. So I eagerly accepted the invitation, intending to present the novel ideas about semantics that I'd brought back with me.

I soon realized, however, that this was an overly ambitious agenda for a one-hour tutorial, particularly if I were to base my presentation on denotational semantics, which seemed highly mathematical and esoteric in those days. (An impression of what I'd like to have presented, given adequate time, can be garnered from Chapters 11 to 14 of Reference [1].) So I took advantage of the close formal similarity between denotational and interpretive semantics to reexpress my presentation in the more familiar context of various interpreters, all written in a purely functional language. Then I discovered that I could classify these interpreters according to whether they used higher-order functions and whether they were independent of the order-of-application of the defining language, and that I could even give constructive (though only informally justified) transformations from higher-level interpreters to more machine-like ones.

The result was "Definitional Interpreters for Higher-Order Programming Languages" [2]. Both as a colloquium at various universities and at the ACM Conference itself (on August 16, 1972) it seemed to make an effective tutorial. (At the conference Art Evans preceded my presentation with a elegant introduction to the lambda calculus and its use in defining his programming language PAL, in which he illustrated both the concepts of syntactic sugar and definition by an interpreter. Unfortunately, the only published record of his talk is an extended abstract [4].)

After preparing the written version of the tutorial, I submitted it to the Communications of the ACM on May 30, 1972, and received three referee reports on

September 27. Although the reports were favorable, they all requested significant revisions; moreover, the editor and one referee felt that the paper should be shortened. I felt it would be impossible to shorten the paper without destroying its tutorial value and limiting its audience to the few people who already understood the technicalities of language definition. Thus I never prepared a revision.

Nevertheless, the conference paper [2] has become a standard source for learning about call-by-value functional languages (i.e., the descendants of Landin's ISWIM [2DI]), the representation of functions by closures, and the continuation-passing-style transformation. Thus, I am delighted by this republication [3], which should make the paper more easily available and more typographically legible. (Citations marked "DI" in this paper refer to the bibliography at the end of Reference [3].)

In the republished version, I have corrected typographical errors, improved the English, and incorporated some of the 26-year-old suggestions of referees and other colleagues. To avoid muddying the historical water, however, I have not corrected the more serious problems that reveal the naïveté of my understanding of semantics in the early 1970's. (Similarly, although I have added bibliographic information about the original references, I have not added references to newer papers.) Instead, I will deal with these matters in the remainder of this introductory note.

1. In Section 4, in the abstract syntax equations

   $$\mathrm{VAL} = \mathrm{INTEGER} \cup \mathrm{BOOLEAN} \cup \mathrm{FUNVAL}$$
   $$\mathrm{FUNVAL} = \mathrm{VAL} \to \mathrm{VAL}$$

   it is not clear what the arrow means. One referee chided me for not invoking Scott's denotational semantics to clarify this meaning, but in fact the equations should make sense regardless of whether the defining language is itself defined denotationally or operationally.

   In retrospect, it is obvious that these equations are a recursive type definition, which presupposes a type system for the defining language. Moreover, FUNVAL is an abstract type, since it describes only those functions from VAL to VAL that denote functional values in the defined language. (In fact, no other functions from VAL to VAL arise in Interpreter I, but that is an accidental consequence of the extreme simplicity of the interpreter.)

2. In the fourth and third paragraphs before the end of Section 5, I should have emphasized the fact that a metacircular interpreter is not really a definition, since it is trivial when the defining language is understood, and otherwise it is ambiguous. In particular, Interpreters I and II say nothing about order of application, while Interpreters I and III say little about higher-order functions.

   Jim Morris put the matter more strongly [5]:

   > The activity of defining features in terms of themselves is highly suspect, especially when they are as subtle as functional objects. It is a fad that should be debunked, in my opinion. A real significance of [a self-defined] interpreter ... is that it displays a simple universal function for the language in question.

On the other hand, I clearly remember that John McCarthy's definition of LISP [1DI], which is a definitional interpreter in the style of II, was a great help when I first learned that language. But it was not the sole support of my understanding.

3. In the fourth paragraph of Section 7, in justifying what is now called the continuation-passing-style transformation, I claimed that "As can be seen with a little thought", the condition

   (A) No operand or declaring expression can cause the application of a serious function.

   implies that

   (B) Whenever some function calls a serious function, the calling function must return the same result as the called function, without performing any further computation.

   Actually, with a little thought, one can see that this claim is false. For example, if s is a serious function, then both

   $$\mathbf{if}\, s(0)\ \mathbf{then}\ 1\ \mathbf{else}\ 2 \qquad \text{and} \qquad s(0)(1)$$

   perform further computation after calling s, yet neither expression contains an operand or declaring expression that causes the application of a serious function.

   It is the converse claim, that B implies A, that is true — and fortunately it is this converse claim that is sufficient for the larger argument: If a program is in "continuation form", this implies B, which implies A, which implies that the meaning of the program is independent of order-of-application.

   In fact, to define a call-by-value language in a way that is independent of the order of application of the defining language, it is not necessary to use a full continuation-passing-style transformation. In Interpreter III, for example, one could replace the sixth line of *eval* by

   $$\begin{aligned} &\mathit{cond?}(r) \rightarrow \mathbf{if}\ \mathit{eval}(\mathit{prem}(r), e, \mathit{mk\text{-}fin}()) \\ &\qquad \mathbf{then}\ \mathit{eval}(\mathit{conc}(r), e, c)\ \mathbf{else}\ \mathit{eval}(\mathit{altr}(r), e, c). \end{aligned}$$

   (Of course, this wouldn't work if the defined language contained **escape** or some other mechanism for introducing continuations as values.)

4. In the third paragraph of Section 8, there is a regrettably obscure description of the connection between definitional interpreters and Scott's denotational semantics. In fact, there are two quite different connections.

   On the one hand, as discussed in the third paragraph, one can use denotational semantics to define the defining language of an interpreter. Specifically, I suggested using denotational semantics to define the defining language of Interpreter IV, which would in turn define a call-by-value language. Such a roundabout method of definition seemed reasonable because I believed that continuations were needed to define a call-by-value language; this belief also motivated my odd remark that "the defining language modelled by Scott uses call by name rather than call by value". Soon after writing "Definitional Interpreters", however, I realized (with a little prodding from Gordon Plotkin) that one can give a direct denotational semantics for a call-by-value language that is almost as straightforward as the direct semantics of a call-by-name language [6].

On the other hand, an entirely different connection is that, for each interpreter in the paper, there is a stylistically similar denotational definition of the defined language of the interpreter. For example, the direct and continuation semantics given in Reference [6] or [1, Sections 11.6 and 12.1] correspond to Interpreters I and IV, while the first-order definition of Reference [1, Section 12.4-12.6] corresponds to Interpreter III.

This stylistic correspondence was both an impetus and a substantial help in actually constructing the various interpreters. Yet I tried to avoid any reference to this correspondence because it obscures the fundamental difference in kind between interpreters and denotational semantics. Nevertheless, a hint of the correspondence surfaces in the claim that Scott models are inherently call-by-name. This provoked a strong reaction from Chris Wadsworth [7]:

> ... Scott treats semantics as being given by a (recursively-defined) mapping from programs to their meanings (values, denotations, etc.) as elements of suitable lattices. Of course, he must have some language in which to write his semantic definitions but, in so far as there is one at all, his defining language is just that of ordinary mathematics. Thus, if, for example, f and x are mathematical expressions for elements, respectively, of a function lattice $[D \to D']$ and of the lattice D, then the function application $f(x)$ unambiguously denotes a certain unique element of the lattice D', and call-by-name/value is irrelevant to this mathematical meaning.

In summary, although the earliest Scott model (D_∞) of a typeless language gave a denotational semantics of a call-by-name language, denotational semantics in general is neutral with respect to call by name versus call by value. Moreover, order of application is an inherently operational concept that is only indirectly pertinent to denotational semantics. (Probably, Wadsworth was especially aware of this fact because he had recently discovered an evaluation method, graph reduction, that is operationally very different from call by name, yet possesses the same denotational semantics [8].)

5. The equivalences between Landin's J-operator [9] and my **escape** operator given in the fourth paragraph of Section 9 are only valid when the expressions asserted to be equivalent occur immediately within a lambda expression (i.e., an abstraction). This limitation has been discussed by Hayo Thielecke [10], who presents a more general solution due to Matthias Felleisen [11].

6. Some of the unpublished work that influenced "Definitional Interpreters" has since been published, and some of the published references have been superceded by more polished and accessible papers. In particular:

(A) An overview of the operational definition of PL/I by the IBM Vienna Laboratory is provided by a journal article [12] that is more accessible than reports such as [17DI].

(B) In 1974, Wadsworth, in collaboration with Christopher Strachey, finally published his work on continuations [13].

(C) In 1993, Michael Fischer published a journal version [14] of his 1972 paper using continuations [27DI].

(D) In 1993, I published a brief history of the discoveries of continuations [15].

In the quarter-century since the appearance of “Definitional Interpreters”, the main topics of the paper have been treated, with far greater rigor, by a variety of authors in a variety of settings. Although a complete review of these developments is far beyond the scope of this paper, the following are a few highlights that I am particularly aware of:

Defunctionalization Although the term “defunctionalization” never took root, the ideas behind it have become commonplace. The concept of a closure, which is a record containing a lambda expression paired with values for the free variables of the expression, first appeared in LISP [16, pages 45 and 133; 17, pages 70–71] (where closures were called “FUNARG triplets”) and in the work of Landin [18, 7DI] (who coined the term “closure”). The replacement of lambda expressions by closure constructors, called “closure conversion”, was used by Guy Steele in designing the RABBIT compiler [19] for Scheme [20]. More recently, closure conversion has been developed in a typed setting by Minamide, Morrisett, and Harper [22].

In “Definitional Interpreters”, however, closures do not contain lambda expressions, but merely unique tags that are in one-to-one correspondence with occurrences of lambda expressions in the program being defunctionalized. The computations described by these occurrences are moved to interpretive functions associated with the points where closures are applied to arguments. Moreover, within each interpretive function the case selection on tags of closures is limited to those tags that might be seen at the point of application.

I've been told that this was an early example of control flow analysis in a functional setting, which has inspired some of the extensive development of this area [23]. In fact, however, the limiting of the case selections was not determined by control flow analysis, but by the informal abstract type declarations (called abstract syntax equations) that guided the construction of the original interpreter.

Continuations In 1974, I proved an appropriate relation between the direct-style and continuation-style denotational semantics of a purely functional language that provided both call by value and call by name [6]. Shortly thereafter, Gordon Plotkin used continuations to give transformations from a call-by-value to a call-by-name language and back, showed that the operational semantics of the transformed programs were independent of order of application, and established simulation relationships between the operational semantics of the untransformed and transformed programs [24]. (He also related his operational semantics, which were in the style of Landin's SECD machine, to a semantics based on reduction.)

In the years since then, further research has established the properties of a variety of continuation-passing-style transformations — in an untyped setting [25], with simple types [26, 27, 28], and with polymorphic types [29]. At the same time, continuations have come to play a major role in the design of compilers for languages with powerful procedure mechanisms [30].

Continuations as values have reappeared with the **catch** operation of Scheme [20]. and the **call/cc** operation of Clinger, Friedman, and Wand [31] (which occurs in modern versions of Scheme [32] and in Standard ML of New Jersey [33]).

Continuations have also been connected with negation in intuitionistic logic (In particular, various continuation-passing-style transformations correspond to double-negation translations of classical into intuitionistic logic [34, 35].)

Effects Nontermination, escapes, and assignment have been recognized by Eugenio Moggi as special cases of a general notion of "effect", which can be defined using the category-theoretic concept of a monad [36]. On the other hand, Andrzej Filinski has shown that a wide variety of Moggi's effects can be defined in terms of escapes (i.e., continuations as values) and assignment [37].

Perhaps the real mystery about these concepts is that they reappear, with easily recognizable similarity, in such a variety of settings: with and without types, in denotational and operational semantics, in interpreters, and even in the transformation of arbitrary programs.

References

1. John C. Reynolds. *Theories of Programming Languages.* Cambridge University Press, Cambridge, England, 1998.
2. John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, volume 2, pages 717–740, Boston, Massachusetts, August 1972. ACM, New York. Reprinted as [3].
3. John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, December 1998.
4. Arthur Evans, Jr. The lambda calculus and its relation to programming languages. In *Proceedings of the ACM Annual Conference*, volume 2, pages 714–716, Boston, Massachusetts, August, 1972. ACM, New York.
5. James H. Morris, Jr. Private communication. September, 1972.
6. John C. Reynolds. On the relation between direct and continuation semantics. In Jacques Loeckx, editor, *Automata, Languages and Programming: 2nd Colloquium*, volume 14 of *Lecture Notes in Computer Science*, pages 141–156, Saarbrücken, Germany, July 29–August 2, 1974. Springer-Verlag, Berlin.
7. Christopher P. Wadsworth. Private communication. August 30, 1972.
8. Christopher P. Wadsworth. *Semantics and Pragmatics of the Lambda-Calculus.* Ph. D. dissertation, Programming Research Group, Oxford University, Oxford, England, September 1971.

9. Peter J. Landin. A generalization of jumps and labels. *Higher-Order and Symbolic Computation*, 11(2):125–143, 1998. Originally a report for UNIVAC Systems Programming Research, dated August 29, 1965.
10. Hayo Thielecke. An introduction to Landin's "A generalization of jumps and labels". *Higher-Order and Symbolic Computation*, 11(2):117–124, 1998.
11. Matthias Felleisen. Reflections on Landin's J-operator: A partly historical note. *Computer Languages*, 12(3/4):197–207, 1987.
12. Peter Lucas and K. Walk. On the formal description of PL/I. *Annual Review in Automatic Programming*, 6(3):105–182, 1969.
13. Christopher Strachey and Christopher P. Wadsworth. Continuations, a mathematical semantics for handling full jumps. Technical Monograph PRG–11, Programming Research Group, Oxford University Computing Laboratory, Oxford, England, January 1974.
14. Michael J. Fischer. Lambda-calculus schemata. *Lisp and Symbolic Computation*, 6(3–4):259–287, November 1993.
15. John C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3–4):233–247, November 1993.
16. John McCarthy, R. Brayton, Daniel J. Edwards, P. Fox, L. Hodes, David C. Luckham, K. Maling, David M. R. Park, and S. Russell. LISP I programmer's manual. Technical report, Computation Center and Research Laboratory of Electronics, Massachusetts Institute of Technology, Cambridge, Massachusetts, March 1, 1960.
17. John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual.* MIT Press, Cambridge, Massachusetts, 1962.
18. Peter J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, January 1964.
19. Guy Lewis Steele Jr. RABBIT: A compiler for SCHEME (a study in compiler optimization). Report AI–TR–474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978.
20. Gerald Jay Sussman and Guy Lewis Steele Jr. SCHEME: An interpreter for extended lambda calculus. AI Memo 349, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, December 1975. Reprinted as [21].
21. Gerald Jay Sussman and Guy Lewis Steele Jr. Scheme: A interpreter for extended lambda calculus. *Higher-Order and Symbolic Computation*, 11(4):405–439, December 1998.
22. Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg Beach, Florida, January 21–24, 1996. ACM Press, New York.
23. Olin Shivers. *Control-Flow Analysis of Higher-Order Languages or Taming Lambda.* Ph. D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1991. Report CMU–CS–91–145.
24. Gordon D. Plotkin. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, December 1975.
25. Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, 6(3–4):289–360, November 1993.
26. Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logic of Programs*, volume 193 of *Lecture Notes in Computer Science*, pages 219–224, Brooklyn, New York, June 17–19, 1985. Springer-Verlag, Berlin.
27. Robert W. Harper, Bruce F. Duba, and David B. MacQueen. Typing first-class continuations in ML. *Journal of Functional Programming*, 3(4):465–484, October 1993.
28. John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *Conference Record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 458–471, Portland, Oregon, January 17–21, 1994. ACM Press, New York.
29. Robert W. Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. *Lisp and Symbolic Computation*, 6(3–4):361–379, November 1993.
30. Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, Cambridge, England, 1992.

31. William Clinger, Daniel P. Friedman, and Mitchell Wand. A scheme for a higher-level semantic algebra. In Maurice Nivat and John C. Reynolds, editors, *Algebraic Methods in Semantics*, pages 237–250. Cambridge University Press, Cambridge, England, 1985.
32. Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(3):7–105, 1998.
33. Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Jan Maluszyński and Martin Wirsing, editors, *Programming Language Implementation and Logic Programming: 3rd International Symposium, PLILP'91*, volume 528 of *Lecture Notes in Computer Science*, pages 1–13, Passau, Germany, August 26–28, 1991. Springer-Verlag, Berlin.
34. Timothy G. Griffin. A formulae-as-types notion of control. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, January 17–19, 1990. ACM Press, New York.
35. Chetan Murthy. *Extracting Constructive Content from Classical Proofs.* Ph. D. dissertation, Department of Computer Science, Cornell University, Ithaca, New York, August 1990. Technical Report 90-1151.
36. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55–92, July 1991.
37. Andrzej Filinski. *Controlling Effects.* Ph. D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania, May 1996. Report CMU–CS–96–119.