State-Transition Machines, Revisited

David A. Schmidt* (schmidt@cis.ksu.edu)
Computing and Information Science Department, Kansas State University, Manhattan, KS 66506 USA

In the autumn of 1978, Neil Jones and Steve Muchnick, working at the University of Kansas, were studying compiler synthesis from Scott-Strachey denotational-semantics definitions; I was Neil's student.

Neil read intently John Reynolds's 1972 paper, *Definitional Interpreters for Higher-Order Programming Languages* [14], and applied Reynolds's continuation-passing and defunctionalization transformations to lambda-calculus-coded denotational-semantics definitions, using the transformed definitions as templates for syntax-directed translation. Neil dubbed the translated source programs, "State-Transition Machines" (STMs), because an object program was a set of equationally defined functions that looked like the transition rules of a finite-state machine.

Our initial efforts were spent on transforming denotational definitions of block-structured, imperative languages into compiling schemes that generated STMs that looked like ordinary assembly code.

In the summer of 1979, Steve moved to the University of California, Berkeley, and Neil and I left for the University of Aarhus, Denmark, where we continued the research project. A summary of the work was eventually published as the paper, *Compiler generation from denotational semantics* [10].

The continuation-passing and defunctionalization transforms were tedious, and I suggested to Neil that one could do better by writing a translator from lambda-calculus into STMs and then constructing a compiler by composing a denotational definition with the lambda-calculus translator. It was unclear whether this tactic would generate better target code than that generated by Neil's smart transformations, but Neil agreed that it was worth a try.

After several false starts, I formulated a translator from a call-by-value lambda-calculus to STMs written in a variant of Landin's SECD-machine, which later appeared in [16] as the "VEC-machine." (Neil preferred a call-by-value lambda-calculus metalanguage.)

At the same time, I was reading Chris Wadsworth's paper, *The relation between computational and denotational properties for Scott's models of the lambda-calculus* [18], and I was fascinated by Wadsworth's use

* Supported by NSF ITR-0085949 and ITR-0086154.

of Böhm trees and head-redex reduction. In particular, Wadsworth's characterization of a lambda-expression as a nested application,

$$(M\, N_0\, N_1 \cdots N_m),\ m \geq 0,$$

suggested an STM-like machine configuration,

$$M\, \langle N_0 : N_1 : \cdots : N_m \rangle,$$

where M was the state name, and $N_0 : N_1 : \cdots : N_m$ was an operand stack. If M was a lambda-abstraction, then head-β-reduction would produce this state transition,

$$(\lambda x.B)\, \langle N_0 : N_1 : \cdots : N_m \rangle \Rightarrow [N_0/x]B\, \langle N_1 : \cdots : N_m \rangle$$

It was easy to see that the substitution of N_0 for x in B could be delayed by means of an environment. All that was missing was a "normalization" rule for $M = (M_0\, M_1)$:

$$(M_0\ M_1)\, \langle N_0 : \cdots : N_m \rangle \Rightarrow M_0\, \langle M_1 : N_0 : \cdots : N_m \rangle$$

The result was a scheme that generated STMs, which I called the "weak-normal-form (WNF) machine." The WNF-machine was in fact the Krivine machine for ordinary β-reduction [3, 4, 11]; it is a standard example of what is now called a *push-enter* machine [12]

The WNF/Krivine machine looked somewhat like a call-by-name SECD-machine less its dump, so for comparison I wrote a simple-minded translation scheme based on the SECD-machine, but the STMs generated by the two schemes were so different in structure that I could not draw any conclusions.

Then, I became curious as to what the continuation-passing and defunctionalization transforms might produce when applied to a denotational definition of the "pure," call-by-name lambda-calculus.[1] I began with Stoy's denotational semantics of the lambda-calculus from his text [17], Chapter 8, and I used Reynolds's continuation-passing semantics from [13]. Thankful that I needed only to defunctionalize, I quickly calculated the result and discovered that it was again the WNF/Krivine machine.

I wrote a summary of my experiments [15], and Neil included it in the proceedings of a workshop he was organizing [8]. Shortly after, Neil used the WNF/Krivine machine to generate STMs of lambda-expressions that he analyzed with iterative data-flow techniques, producing one of the first closure analyses [9].

[1] To this point, we had worked with an applied lambda-calculus meta-language, essentially the one in Reynolds's paper [14].

In retrospect, it is fair to say that Neil had the main insights and that I derived the WNF/Krivine machine as an isolated exercise. For example, I never considered applying systematically the continuation-conversion and defunctionalization transforms to the family of lambda-calculi variations; Ager et al. [1] have since done so and derived not only the Krivine machine but Felleisen et al.'s CEK machine [6], Hannan and Miller's CLS machine [7], and the Categorical Abstract Machine [2]. And Danvy showed how to travel in the reverse direction, mapping the SECD machine to its corresponding evaluation function [5].

Years later, I was happy to see the WNF machine popularized as the Krivine machine [3, 4, 11] — I was always impressed that the supposedly difficult-to-implement, leftmost-outermost, call-by-name reduction strategy had such a simple interpretive formulation.

I would like to thank Mads Sig Ager and Olivier Danvy for rereading my paper from 1979, and I would like to thank Olivier in particular for proposing that I revise it for this special issue of *HOSC*. I have corrected some small errors, cleaned the narrative, and deleted a well-intended but misleading section on a call-by-value, WNF-machine variant.

Acknowledgements

Olivier Danvy's comments on a draft of this note are most appreciated.

References

1. Ager, M., D. Biernacki, O. Danvy, and J. Midtgaard': 2003, 'A functional correspondence between evaluators and abstract machines'. In: *Proc. 5th ACM-SIGPLAN Int. Conf. Principles and Practice of Declarative Programming*. pp. 8–19.
2. Cousineau, G. and P.-L. Curien: 1987, 'The categorical abstract machine'. *Science of Computer Programming* **8**, 173–202.
3. Crégut, P.: 1990, 'An abstract machine for lambda-terms normalization'. In: *Proc. ACM Symp. Lisp and Functional Programming*. pp. 333–340.
4. Curien, P.-L.: 1991, 'An abstract framework for environment machines'. *Theoretical Computer Science* **82**, 389–402.
5. Danvy, O.: 2004, 'A Rational Deconstruction of Landin's SECD Machine'. In: *16th Int. Workshop, Implementation and Application of Functional Languages*. pp. 52–71. Extended version available as the technical report BRICS RS-03-33.
6. Felleisen, M. and D. Friedman: 1986, 'Control operators, the SECD-machine, and the λ-calculus'. In: M. Wirsing (ed.): *Formal Description of Programming Concepts III*. North-Holland, Amsterdam, pp. 193–217.
7. Hannan, J. and D. Miller: 1992, 'From operational semantics to abstract machines'. *Math. Structures in Computer Science* **2**, 415–459.

8. Jones, N. (ed.): 1980, *Semantics-Directed Compiler Generation*. Lecture Notes in Computer Science 94, Springer-Verlag.
9. Jones, N.: 1981, 'Flow analysis of lambda expressions'. In: *Int. Conf. on Automata, Languages, and Programming*. Lecture Notes in Computer Science 115, Springer-Verlag, pp. 114–128.
10. Jones, N. and D. Schmidt: 1980, 'Compiler generation from denotational semantics'. In: N. Jones (ed.): *Semantics-Directed Compiler Generation*. Lecture Notes in Computer Science 94, Springer-Verlag, pp. 70–93.
11. Krivine, J.-L.: 1985, 'Un interprète du λ-calcul'. Brouillon. Available online at `http://www.pps.jussieu.fr/~krivine/`.
12. Peyton Jones, S.: 1992, 'Implementing lazy functional languages on stock hardware: the spineless, tagless G-machine'. *J. Functional Programming* **2**, 127–202.
13. Reynolds, J.: 1974, 'On the relation between direct and continuation semantics'. In: *Proc. 2d Colloquium on Automata, Languages, and Programming*. Lecture Notes in Computer Science, Springer-Verlag, pp. 141–156.
14. Reynolds, J. C.: 1998, 'Definitional Interpreters for Higher-Order Programming Languages'. *Higher-Order and Symbolic Computation* **11**(4), 363–397. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword.
15. Schmidt, D.: 1980, 'State transition machines for lambda-calculus expressions'. In: N. Jones (ed.): *Semantics-Directed Compiler Generation*. Lecture Notes in Computer Science 94, Springer-Verlag, pp. 415–440.
16. Schmidt, D.: 1986, *Denotational Semantics*. Allyn and Bacon, Boston.
17. Stoy, J.: 1977, *Denotational Semantics*. MIT Press, Cambridge, MA.
18. Wadsworth, C.: 1976, 'The relation between computational and denotational properties for Scott's models of the lambda-calculus'. *SIAM J. of Computing* **5**, 488–521.