Preface

During the second half of the twentieth century, computing and computation theory were established as part of a common academic discipline. One distinctive aspect of computation theory is the importance of its notations: Programming languages like Algol 60, Scheme, and Smalltalk are significant and studied in their own right, because they provide the means for expressing the very *essence of computation*. This "essence" is formalized within the subareas of computational complexity, program design and analysis, and program transformation.

This volume presents state-of-the-art aspects of the essence of computation — computational complexity, program analysis, and program transformation — by means of research essays and surveys written by recognized experts in the three areas.

An Appreciation of Neil D. Jones

Aside from the technical orientation of their writings, a thread that connects all the authors of this volume is that all of them have worked with *Prof. Neil D. Jones*, in the roles of student, colleague, and, in one case, mentor. The volume's authors wish to dedicate their articles to Neil on the occasion of his 60th birthday.

The evolution of Neil Jones's career parallels the evolution of computation theory itself; indeed, inspiration and influence from Neil is evident throughout the area he studied and helped shape. The following survey (and reference list) of Neil's work is necessarily selective and incomplete but will nonetheless give the reader a valid impression of Neil's influence on his field.

Born on March 22, 1941, in Centralia, Illinois, USA, Neil studied first at Southern Illinois University (1959–1962) and then at the University of Western Ontario, Canada (1962–1967), where he worked first under the supervision of Satoru Takasu (who later led the Research Institute of Mathematical Sciences at Kyoto) and then under Arto Salomaa (now at the University of Turku). During this period, Neil led the development of one of the first, if not the very first, Algol 60 compilers in North America [5, 34]. The problems and issues raised by implementing perhaps the quintessential programming language would stimulate Neil for decades to come.

During his tenure as a faculty member at Pennsylvania State University (1967–1973), Neil established himself as an expert in formal language theory and computability, distinguishing himself with his breakthrough solution to the "spectrum" problem, jointly with Alan Selman [6, 7, 36]. Appreciating the significance of the fast-growing academic discipline of computing science, Neil authored the first computation-theory text specifically oriented towards computing scientists, *Computability Theory: An Introduction* [8], which was one of the first volumes published in the newly established ACM Monograph series.

Neil's scholarly work at Pennsylvania State laid the foundation for his move to the University of Kansas in 1973, where Neil provided a crucial push to the

blossoming area of computational complexity: His seminal research papers [9, 23, 24] introduced and developed the complexity classes of polynomial time and log space, and they defined and applied the now classic technique of log-space reduction for relating computational problems.

Neil's continuing interest in programming languages influenced his research in computational complexity, which acquired the innovative aspect of exploiting the notations used to code algorithms in the proofs of the complexity properties of the algorithms themselves. Key instances of these efforts were published in collaboration with Steven Muchnick at the University of Kansas [26, 27].

Around 1978, programming-language analysis moved to the forefront of Neil's research. Neil Jones was perhaps the first computing scientist to realize the crucial role that *binding times* play in the definition and implementation of a programming language: Early binding times support detailed compiler analysis and fast target code, whereas late binding times promote dynamic behavior of control and data structure and end-user freedom. His text with Steven Muchnick, *Tempo: A Unified Treatment of Binding Time and Parameter Passing Concepts* [28], remains to this day the standard reference on binding-time concepts.

Implicit in the *Tempo* text was the notion that a program could be mechanically analyzed to extract the same forms of information that are communicated by early binding times. This intuition led Neil to develop several innovative data-flow analysis techniques [11, 25, 29–32], coedit a landmark collection of seminal papers on flow analysis and program analysis, *Program Flow Analysis: Theory and Applications* [43], and coauthor an influential survey paper [33].

A deep understanding of compiler implementation, coupled with many insights into binding times and flow analysis, stimulated Neil to explore aspects of compiler generation from formal descriptions of programming language syntax and semantics. While visiting Aarhus University, Denmark, in 1976-77, Neil encountered the just developed denotational-semantics format, which served as a testbed for his subsequent investigations into compiler generation. His own efforts, e.g., [35, 39, 42], plus those of his colleagues, as documented in the proceedings of a crucial workshop organized by Neil in 1980 [10], display a literal explosion of innovative techniques that continue to influence compiler generation research.

In 1981, Neil accepted a faculty position at the University of Copenhagen, where he was promoted to Professor. His research at Copenhagen was a culmination of the research lines followed earlier in his career: Building on his knowledge in complexity and program analysis, Neil developed a theory of program transformation, which now constitutes a cornerstone of the field of *partial evaluation.*

The story goes somewhat like this: Dines Bjørner introduced Neil to Andrei Ershov's theory of "mixed computation" — a theory of compilation and compiler generation stated in terms of interpreter self-interpretation (self-application). Intrigued by the theory's ramifications, Neil and his research team in Copenhagen worked on the problem and within a year developed the first program — a *partial evaluator* — that generated from itself a compiler generator by means of self-application [37]. The crucial concept needed to bring Ershov's theory to life

was a flow analysis for calculating binding times, a concept that only Neil was perfectly posed to understand and apply!

The programming of the first self-applicable partial evaluator "broke the dam," and the flood of results that followed showed the practical consequences of partial-evaluation-based program transformation [2–4, 12–16, 20, 22, 38, 44]. Along with Bjørner and Ershov, Neil co-organized a partial-evaluation workshop in Gammel Avernæs, Denmark, in October 1987. The meeting brought together the leading minds in program transformation and partial evaluation and was perhaps the first truly *international* computing science meeting, due to the presence of a significant contingent of seven top researchers from Brezhnev's USSR [1].

Neil coauthored the standard reference on partial evaluation in 1993 [21].

Coming full circle at the end of the century, Neil returned to computational complexity theory, blending it with his programming languages research by defining a family of Lisp-like mini-languages that characterize the basic complexity classes [18, 19]. His breakthroughs are summarized in his formidable text, *Computability and Complexity from a Programming Perspective* [17], which displays a distinctive blend of insights from complexity, program analysis, and transformation, which could have come only from a person who has devoted a career to deeply understanding all three areas.

If his recent work is any indication [40, 41], Neil's investigations into the *essence of computation* are far from finished!

15 October 2002

Torben Mogensen
David Schmidt
I. Hal Sudborough

References

1. Dines Bjørner, Andrei P. Ershov, and Neil D. Jones, editors. *Partial Evaluation and Mixed Computation*. Elsevier Science/North-Holland, 1988.
2. Harald Ganzinger and Neil D. Jones, editors. *Programs as Data Objects*. Lecture Notes in Computer Science 217. Springer-Verlag, 1986.
3. Arne John Glenstrup and Neil D. Jones. BTA algorithms to ensure termination of off-line partial evaluation. In *Perspectives of System Informatics: Proceedings of the Andrei Ershov Second International Memorial Conference*, Lecture Notes in Computer Science 1181. Springer-Verlag, 1996.
4. Carsten K. Gomard and Neil D. Jones. A partial evaluator for the untyped lambda calculus. *Journal of Functional Programming* 1(1):21-69, January 1991.
5. Neil D. Jones. 1620 Algol compiler: subroutine package. Technical Report. Southern Illinois University, Computing Center, 1962.
6. Neil D. Jones. Classes of automata and transitive closure. *Information and Control* 13:207-209, 1968.
7. Neil D. Jones. Context-free languages and rudimentary attributes. *Mathematical Systems Theory* 3(2):102-109, 1969. Corrigendum, 11:379-380.

8. Neil D. Jones. *Computability Theory: An Introduction.* ACM Monograph Series. Academic Press, 1973.
9. Neil D. Jones. Space-bounded reducibility among combinatorial problems. *Journal of Computer and System Science* 11:68-85, 1975.
10. Neil D. Jones, editor. *Semantics-Directed Compiler Generation.* Lecture Notes in Computer Science 94. Springer-Verlag, 1980.
11. Neil D. Jones. Flow analysis of lambda expressions. In *Automata Languages and Programming*, Lecture Notes in Computer Science 115. Springer-Verlag, 1981, pp. 114-128.
12. Neil D. Jones. Partial evaluation, self-application and types. In *Automata, Languages and Programming*, Lecture Notes in Computer Science 443. Springer-Verlag, 1990, pp. 639-659.
13. Neil D. Jones. Static semantics, types and binding time analysis. *Theoretical Computer Science* 90:95-118, 1991.
14. Neil D. Jones. The essence of program transformation by partial evaluation and driving. In *Logic, Language and Computation, a Festschrift in Honor of Satoru Takasu,* Masahiko Sato, Neil D. Jones, and Masami Hagiya, editors, Lecture Notes in Computer Science 792. Springer-Verlag, 1994, pp. 206-224.
15. Neil D. Jones. MIX ten years later. In William L. Scherlis, editor, *Proceedings of PEPM '95.* ACM Press, 1995, pp. 24-38.
16. Neil D. Jones. What *not* to do when writing an interpreter for specialisation. In *Partial Evaluation*, Olivier Danvy, Robert Glück, and Peter Thiemann, editors, Lecture Notes in Computer Science 1110. Springer-Verlag, 1996, pp. 216-237.
17. Neil D. Jones. *Computability and Complexity from a Programming Perspective.* MIT Press, London, 1997.
18. Neil D. Jones. LOGSPACE and PTIME characterized by programming languages. *Journal of Theoretical Computer Science* 228:151-174, 1999.
19. Neil D. Jones. The expressive power of higher-order types. *Journal of Functional Programming* 11(1):55-94, 2001.
20. Neil D. Jones, Carsten K. Gomard, Anders Bondorf, Olivier Danvy, and Torben Mogensen. A self-applicable partial evaluator for the lambda calculus. In *International Conference on Computer Languages.* IEEE Computer Society, 1990, pp. 49-58.
21. Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice-Hall, 1993.
22. Neil D. Jones, Masami Hagiya, and Masahiko Sato, editors. *Logic, Language and Computation, a Festschrift in Honor of Satoru Takasu.* Lecture Notes in Computer Science 792. Springer-Verlag, 1994. 269 pages.
23. Neil D. Jones and W.T. Laaser. Problems complete for deterministic polynomial time. *Journal of Theoretical Computer Science* 3:105-117, 1977.
24. Neil D. Jones and E. Lien. New problems complete for nondeterministic log space. *Mathematical Systems Theory* 10(1):1-17, 1976.
25. Neil D. Jones and Steven Muchnick. Automatic optimization of binding times. In *Proceedings of the Symposium on Principles of Programming Languages*, Vol. 3, 1976, pp. 77-94.
26. Neil D. Jones and Steven Muchnick. Even simple programs are hard to analyze. *Journal of the ACM* 24(2):338-350, 1977.
27. Neil D. Jones and Steven Muchnick. Complexity of finite memory programs with recursion. *Journal of the ACM* 25(2):312-321, 1978.

28. Neil D. Jones and Steven Muchnick. *Tempo: A Unified Treatment of Binding Time and Parameter Passing Concepts*. Lecture Notes in Computer Science 66. Springer-Verlag, 1978.
29. Neil D. Jones and Steven Muchnick. Flow analysis and optimization of Lisp-like structures. In *Proceedings of the Symposium on Principles of Programming Languages*, Vol. 6, 1979, pp. 244-256.
30. Neil D. Jones and Steven Muchnick. A flexible approach to interprocedural data flow analysis. In *Proceedings of the Symposium on Principles of Programming Languages*, Vol. 9, 1982, pp. 66-94.
31. Neil D. Jones and Alan Mycroft. Data flow analysis of applicative programs using minimal function graphs. In *Proceedings of the Symposium on Principles of Programming Languages*, Vol. 13, 1986, pp. 296-306.
32. Neil D. Jones and Alan Mycroft. A relational approach to program flow analysis. In *Programs as Data Objects,* Harald Ganzinger and Neil D. Jones, editors, Lecture Notes in Computer Science 217. Springer-Verlag, 1986, pp. 112-135.
33. Neil D. Jones and Flemming Nielson. Abstract interpretation: a semantics-based tool for program analysis. In *Handbook of Logic in Computer Science*. Oxford University Press, 1994, pp. 527-629.
34. Neil D. Jones, P. Russel, A.G. Wilford, and W. Wilson. IBM 7040 Algol compiler. Technical Report. University of Western Ontario, Computer Science Department, 1966.
35. Neil D. Jones and David Schmidt. Compiler generation from denotational semantics. In *Semantics-Directed Compiler Generation*, Neil D. Jones, editor, Lecture Notes in Computer Science 94. Springer-Verlag, 1980, pp. 70-93.
36. Neil D. Jones and Alan L. Selman. Turing machines and the spectra of first order formula with equality. *Journal of Symbolic Logic* 39(1):139-150, 1974.
37. Neil D. Jones, Peter Sestoft, and Harald Søndergaard. An experiment in partial evaluation: the generation of a compiler generator. In *Conference on Rewriting Techniques and Applications,* J.-P. Jouannoud, editor, Lecture Notes in Computer Science 202. Springer Verlag, 1985, pp. 125-140.
38. Neil D. Jones, Peter Sestoft, and Harald Søndergaard. Mix: a self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1): 9-50, 1989.
39. Neil D. Jones and Mads Tofte. Towards a theory of compiler generation. In *Proceedings of the Workshop on Formal Software Development Methods,* D. Bjorner, editor. North-Holland, 1986.
40. David Lacey, Neil D. Jones, Eric Van Wyck, and Carl Christian Frederiksen. Proving correctness of compiler optimizations by temporal logic. In *ACM Symposium on Principles of Programming Languages*, Vol. 29, 2002, pp. 283-294.
41. Chin S. Lee, Neil D. Jones, and Amir Ben-Amran. The size-change principle for program termination. In *ACM Symposium on Principles of Programming Languages*, Vol. 28. ACM Press, 2001, pp. 81-92.
42. Kim Marriot, Harald Søndergaard, and Neil D. Jones. Denotational abstract interpretation of logic programs. *ACM Transactions on Programming Languages and Systems* 16(3): 607-648, 1994.
43. Steven Muchnick and Neil D. Jones, editors. *Program Flow Analysis: Theory and Applications.* Prentice-Hall, 1981.
44. Morten Heine Sørensen, Robert Glück, and Neil D. Jones. Towards unifying deforestation, supercompilation, partial evaluation, and generalized partial computation. In *European Symposium on Programming*, D. Sannella, editor, Lecture Notes in Computer Science 788. Springer-Verlag, 1994, pp. 485-500.