ACM SIGPLAN
Influential Papers

Editor: *Fermín Reig, Univ. of Nottingham and Michael Franz, Univ. of California, Irvine; f.reig@cs.nott.ac.uk, franz@uci.edu*

Reminiscences on Influential Papers

This new column is about the dissemination of scientific discovery and how this influences our research, often in unpredictable ways.

We have asked a few well-known and respected people in the programming language community to identify a paper that had a major influence on their research, and to describe what they liked about that paper and the impact it had on them. Their responses make very interesting reading.

This SIGPLAN Notices column follows the format of a very popular column in ACM's SIGMOD Record. We are grateful to the editors of that column for letting us borrow their idea and title.

Philip Wadler, University of Edinburgh, `wadler@inf.ed.ac.uk`.

[John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM National Conference*, pages 717–740, August 1972.]

Editor's note: This paper was reprinted in *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. See also 'Definitional Interpreters Revisited', in the same issue (pages 355–361).

Certain papers change your life. McCarthy's 'Recursive Functions of Symbolic Expressions and their Computation by Machine (Part I)' (1960) changed mine, and so did Landin's 'The Next 700 Programming Languages' (1966). And I remember the moment, halfway through my graduate career, when Guy Steele handed me Reynolds's 'Definitional Interpreters for Higher-Order Programming Languages' (1972).

It is now common to explicate the structure of a programming language by presenting an interpreter for that language. If the language interpreted is the same as the language doing the interpreting, the interpreter is called *meta-circular*. Interpreters may be written at differing levels of detail, to explicate different implementation strategies. For instance, the interpreter may be written in a *continuation-passing style*; or some of the higher-order functions may be represented explicitly using data-structures, via *defunctionalisation*. More elaborate interpreters may be derived from simpler versions, thus providing a methodology for discovering an implementation strategy and showing it correct. Each of these techniques has become a mainstay of the study of programming languages, and all of them were introduced in this single paper by Reynolds.

The paper had a profound effect on me because it placed an elegant, logical structure on so much of what had come before. McCarthy's interpreter for Lisp in Lisp and Landin's SECD machine were well and good, but they contained "cogs and wheels" whose origin appeared arbitrary. I was particularly concerned that Landin's J operator in Iswim (a forerunner of call/cc in Scheme) seemed to depend crucially on what I thought were arbitrary aspects of the SECD machine. Reynolds's starting point was the simplest possible interpreter for lambda calculus in lambda calculus, with no room for any arbitrary choice, not even that of evaluation order. From this he derived, by simple and obviously correct steps of program transformation, interpreters that were essentially equivalent to McCarthy's and Landin's. In the process, among other things, he gave an elegant resolution of the "FUNARG" question of Lisp, and gave the first explanation of the J operator that I felt made sense. His penetrating and systematic development stuck with me as an epitome to aim for in all my work: to use theory to clarify practice, and practice to inspire theory.

Krzysztof R. Apt, CWI (currently at the National University of Singapore), `apt@cwi.nl`.

[O.-J. Dahl, E.W. Dijkstra and C.A.R. Hoare. *Structured Programming.* Academic Press, 1972.]

I came to computer science from mathematical logic. In 1975 I met Edsger Dijkstra when I knew nothing about computer science. He recommended me to study the book "Structured Programming" by Dahl, Dijkstra, and Hoare that contained three articles written by the authors. Studying them allowed me to appreciate the beauty of the

programs and of the programming languages and raised my interest in program verification. The article of Dijkstra explained how programs can be systematically developed. The second article, by Hoare, provided the theoretical background for the type system of Pascal. The third article, by Dahl and Hoare discussed the benefits of the class concept as realized in the Simula language. After 31 years, the book is still on sale, see `amazon.com`, and still worthwhile to read.

Each of the authors received eventually the ACM Turing Award.

Matthias Felleisen, Northeastern University.

[Gordon Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.]

On the very last day of my first graduate course (December 1984), my PhD advisor (Daniel Friedman, Indiana) handed out Plotkin's paper and said something like "I think this is interesting but I haven't really absorbed it yet." The next day I flew back to Germany for Christmas and read the paper on the airplane. Needless to say, I didn't understand a word beyond the introduction and put it away without a second thought. Six months later, Dan and I were trying to figure out how to create a lambda calculus with a Scheme-inspired call/cc operator. Eventually, I recognized the connection between our idea and the introduction to Plotkin's paper. So, I spent the next two months on the paper, using Barendregt's book as an auxiliary reading.

Plotkin's paper consists of two parts. The first part explains five aspects of an operational model of a programming language: the abstract (virtual) machine; the calculus (a "proof" system); the reduction system (another "proof" system); the standard reduction relation (yet another one); and its observational equivalence relation ("truth"). It does so with two small examples: a functional language based on the call-by-name lambda calculus and another one based on the call-by-value calculus. By designing a calculus for a functional call-by-value language, Plotkin clarifies that the conventional lambda call-by-name calculus does not play a canonical role in programming languages. Instead, when we wish to study a programming language, we should investigate its operational equivalence and design sound calculi and reduction systems.

The second part of the paper is about the relationship between the by-name and by-value languages. Plotkin proves that the continuation-passing transformation maps the call-by-value calculus in a sound but incomplete manner to the call-by-name calculus; he also poses a number of research questions about this relationships.

As I studied the paper, I began to see the first part of the paper as not just a result but an implicit research program. Starting with my dissertation research, I spent many years working out a new style of semantics for programming languages (reductions based on evaluation contexts), showing its applicability to a reasonably broad spectrum of languages, and validating the usefulness of this style for many situations (with Friedman, Hieb, Wright). I also worked on some of the open questions in the second part of the paper (with Flanagan, Sabry, Sitaram), though by no means all of them. Because of my own personal success with Plotkin's paper, I have recommended it to many people over the past twenty years, and I am happy to see that it has found a large audience. The paper has also set my standards for research papers—it is not just about what a paper says but what it inspires. By those standards, Plotkin has done extremely well here.