RETROSPECTIVE:

Higher-order control-flow analysis in retrospect: Lessons learned, lessons abandoned

Olin Shivers

College of Computing
Georgia Institute of Technology
Atlanta, Georgia, USA
shivers@cc.gatech.edu

Introduction

A tremendous amount of work has been carried out in the general area of program analysis of higher-order languages, both before and after the publication of “Control-flow analysis in Scheme” in PLDI. In the space allowed here, I could never hope to tie my 1988 paper properly into this daunting canon of research. So, instead, I will focus on summarising some major directions that were pursued afterwards, and (more significantly) some elements of the paper’s message that, to me, seem to have been abandoned.

Extensions and limitations

“Control-flow analysis in Scheme” introduced the notion of 0CFA and the k-CFA hierarchy (though the general idea of control-flow analysis for functional languages has multiple precedents [8]). It did not take long to discover that the basic analysis, for any $k > 0$, was intractably slow for large programs. In the ensuing years, researchers have expended a great deal of effort deriving clever ways to tame the cost of the analysis. Three examples from this vein of work are:

- Wright and Jagannathan [21] exploited the “hint” of type polymorphism to restrict or focus the polyvariance of the analysis to cases where its greater discrimination would be more likely to pay off.
- Heintze and McAllester developed an algorithm that exploits static type information to implement an on-demand analysis [7], providing gains in sparse applications where a program-manipulation system asks relatively few control-flow questions of the program being analysed.
- Ashley used Cousot’s notion of a “widening” lattice operator to coarsen the analysis for speedups [2].

Other researchers developed general frameworks for expressing CFA in the higher-order setting. Here, the *ur*-framework is the Cousots’ non-standard abstract interpretation [3, 4], which I used in later work. The Nielsons were also instrumental in formalising a general framework for CFA; the text *Principles of Program Analysis* [13], written with Hankin, summarises much of this work.

However, despite all this work on formalising CFA and speeding it up, I have been disappointed in the dearth of work extending its *power.* “Control-flow analysis in Scheme” delineates a fundamental barrier beyond which the whole k-CFA framework cannot go, which I called the “environment problem:” some analyses are rendered unsound by the inevitable folding of the infinite set of precise environments down to a tractably finite set.

I provided a solution to this problem, which I called “reflow analysis,” in my subsequent doctoral dissertation [17]. This is a thread that has not been picked up—a lesson abandoned. Reflow analysis needs more investigation. It has not been given, in my dissertation or elsewhere, the kind of detailed formalisation that 0CFA enjoys today. This is a pity, since there are interesting program analyses and optimisations that require this kind of extra power.

CPS as an intermediate representation

It was fundamental to my development of control-flow analysis to use an intermediate representation based on continuation-passing style (CPS). This continued a line of work originating with Steele’s seminal Rabbit compiler [18], and continuing through Orbit [11], Kelsey’s transformational compiler TC [9], and SML/NJ, whose development was led by Appel and MacQueen [1]. Kelsey subsequently pointed out that the SSA revolution in the traditional compiler community was essentially CPS in another guise [10].

The use of CPS drastically simplified my analyses by replacing the full gamut of control constructs—iteration, branching, function call, function return, sequencing, and non-local transfers—with a single mechanism, the tail-recursive procedure call. In CPS, the entire control-flow problem reduces simply to discovering which call sites invoke which lambdas.

In the early 90’s, however, a series of influential papers by Sabry, *et al.* [15, 6] made the case for abandoning CPS in favor of alternate, “direct-style” intermediate representations, such as A-normal form. These representations have now become standard practice.

In 2002, then, CPS would appear to be a lesson abandoned. I would argue that the use of CPS remains a boon to program analysis. Not only does CPS provide a uniform representation of control structure, it also packages up *evaluation context* into a familiar form: lambda. A functional-program analysis system already has powerful machinery for reasoning about lambdas; the CPS transform allows this machinery to be employed to reason about context, as well. Without CPS, separate contextual analyses and transforms must be also implemented—redundantly, in my view.

In the fourteen years that have passed since the publication of “Control-flow analysis in Scheme,” I have seen many developments of k-CFA variants done in a direct-style setting. They are all, to my eye, needlessly complicated by the profusion of control points and control mechanisms made necessary by direct style—harder to develop, harder to understand.

20 Years of the ACM/SIGPLAN Conference on Programming Language Design and Implementation (1979-1999): A Selection, 2003.

Denotational and operational semantics

When I formalised 0CFA in subsequent work, I did so as a non-standard abstract semantics using a denotational approach. Why not use operational semantics, which more naturally exposes the intermediate states of the abstract machine for collection? The reason is not profound: denotational semantics was what I knew, and I had access, as a graduate student at CMU, to a vast pool of local expertise on the subject.

At almost the exact same time that I was developing k-CFA using denotational semantics, Deutsch, for example, was attacking pointer analyses in higher-order languages—an extremely difficult and complex problem—with great success using an operational formalisation [5]. As another example, Wand and his students have done CFA-based analyses and transforms in the operational setting [19].

For a CPS language, a collecting semantics in a denotational framework is not greatly different from a big-step operational semantics, so the choice of approach is not that critical. A small-step semantics, however, does have distinct technical differences that are worth considering.

Weaving the strands together

There are many analytic tools we can use to reason about programs: data-flow analyses, type systems, semantics, both precise and abstract, and proof systems to reason about them. 0CFA occupies one point in this complex space; it is able to tell particular kinds of stories about a particular kind of program behaviour.

Perhaps the most exciting development in the program-analysis field in recent years is the emergence of a coherent structure relating these different approaches. As Palsberg [14] and the Church group [20] have shown, control flow can be captured with flow types and type analyses. Types themselves merge into full-blown verification logics by way of the Curry-Howard isomorphism; this shows up practically in the proof systems used in proof-carrying code [12]. Data-flow analyses can be seen simply as particular proof-generating strategies; Schmidt has connected them to temporal logics and operational semantics via abstract interpretation [16].

References

[1] Andrew W. Appel. *Compiling with Continuations.* Cambridge University Press, 1992.

[2] J. Michael Ashley and R. Kent Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Transactions on Programming Languages and Systems* 20(4), pages 845–868, July 1988.

[3] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[4] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 269–282, 1979.

[5] Alain Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications (extended version). Research Report LIX/RR/90/11, LIX, Ecole Polytechnique, 91128 Palaiseau Cedex, France.

[6] Cormac Flanagan, Amr Sabry, Bruce Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 237–247, 1993.

[7] Nevin Heintze and David McAllester. Linear-time subtransitive control flow analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1997.

[8] Neil D. Jones. Flow analysis of lambda expressions. In *Automata Languages and Programming*, volume 115, pages 114–128, Lecture Notes in Computer Science, Springer-Verlag, 1981.

[9] Richard A. Kelsey. *Compilation by Program Transformation.* Ph.D. dissertation, Yale University, May 1989. Research Report 702, Department of Computer Science.

[10] Richard A. Kelsey. A correspondence between continuation-passing style and static single assignment form. In *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations, SIGPLAN Notices* 30(3), pages 13–22, 1995.

[11] David Kranz. ORBIT*: An Optimizing Compiler for Scheme.* Ph.D. dissertation, Yale University, February 1988. Research Report 632, Department of Computer Science.

[12] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Proceedings of the Second Symposium on Operating Systems Design and Implementation* (OSDI'96), pages 229–243, 1996.

[13] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis.* Springer, 1999.

[14] Jens Palsberg and Patrick M. O'Keefe. A type system equivalent to flow analysis. *ACM Transactions on Programming Languages and Systems*, 17(4), pages 576–599, July 1995.

[15] Amr Sabry and Matthias Felleisen. Is continuation passing useful for data-flow analysis? In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–12, 1994.

[16] David Schmidt. Data-flow analysis is model checking of abstract interpretations. In *Proceedings of the 25th ACM Symposium on Principles of Programming Languages* (POPL '97), pages 38–48, 1998.

[17] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* Ph.D. dissertation, Carnegie Mellon University, May 1991. Technical Report CMU-CS-91-145, School of Computer Science.

[18] Guy L. Steele Jr. RABBIT*: A Compiler for* SCHEME. Technical Report 474, MIT AI Lab, May 1978.

[19] Mitchell Wand and Igor Siveroni. Constraint systems for useless variable elimination. In *Proceedings of the 26th ACM Symposium on Principles of Programming Languages*, pages 291–302, 1999.

[20] J. B. Wells, Allyn Dimock, Robert Muller, and Franklyn Turbak. A calculus for polymorphic and polyvariant flow types. *Journal of Functional Programming. (To appear.)*

[21] Andrew K. Wright and Suresh Jagannathan. Polymorphic splitting: An effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems* 20(1), pages 166–207, January 1998.