Compilers for higher-order languages often transform programs into continuation-passing style (CPS) form as an intermediate representation. In the language of CPS terms, all procedures take a continuation argument k that represents the "rest of the computation;" procedures apply k to their return value instead of returning directly. This form is beneficial to compilers because it enables optimizations by way of simple β- and η-reductions, and it simplifies code generation by resembling a low-level target language.

While implementing a naïve CPS transformation is a straightforward task, for realistic compilers the process typically involves at least two additional phases. First, the CPS transformation introduces a large number of "administrative" terms which greatly increase a program's size. A simplification phase reduces the resulting programs to a smaller, normal form. Second, compilers treat the continuation closures specially, e.g., to improve analyses or provide better allocation strategies.

Flanagan et al. [2] argue that by using a simpler organization based on the A-reductions of Sabry and Felleisen [3], compilers could achieve the benefits of a fully developed CPS compiler with a single source-level transformation. In support of their argument they present a sequence of abstract machines simulating the compilation of a Scheme-like language. They begin by deriving the CEK machine from the CE machine specialized for (simplified) CPS, showing how to use the continuation register to model the optimizations of realistic CPS compilers. Next they introduce the A-reductions and the corresponding CEK machine, which they prove is equivalent to the CEK machine for CPS. In fact, the machines are identical up to the syntax of their control strings, and applying an inverse CPS transformation to a simplified CPS term gives the A-normal form of the original source-language term. They conclude that compiling with A-normal forms realizes the essence of compiling with continuations, without requiring a multi-stage CPS transformation.

While we understood Flanagan et al.'s reasoning at a high level after an initial reading, we planned to try and grasp the subtleties of the abstract machines and language transformations by implementing them in Redex. Specifically, we were interested in better understanding how programs change throughout the transformations and how redundancies are removed from the CEK machine for CPS. We also wanted to implement the A-normalization algorithm as a reduction relation using the evaluation contexts defined in the paper, whose presentation appeared puzzling to us at first.

We transcribed the figures in the paper into language definitions, metafunctions, and reduction relations in Redex. By running the transformations on concrete examples, we gained more of an intuition for how they work. After reading over the CEK machines a second time in order to implement them, we were able to see the redundancies the authors discuss—how the machine for CPS ignores the explicit continuation closures and instead manipulates the continuation register. For the A-normalization algorithm, we included additional rules to allow reducible expressions to appear in the bodies of **let** and **if0** terms, for example, since these cases weren't

explicitly covered by the evaluation contexts defined in the paper. Alternatively, we could have defined a metafunction according to the code in the paper's appendix.

As a side effect, this project also helped us discover how a tool such as Redex is useful to programming language researchers in the real world. For example, as we began implementing a model of the paper, we noticed two small typos. In particular, for the naïve transformation (where $\mathcal{F}$ is due to Fischer [1]), $\mathcal{F}[\![M]\!]$ should be $(\mathcal{F}[\![M]\!]\ k)$ in $\Phi[\![\lambda x_1 \ldots x_n.M]\!] = \lambda k x_1 \ldots x_n.\mathcal{F}[\![M]\!]$; and for the inverse transformation, $\mathcal{U}[\![M]\!]$ should be $\mathcal{U}[\![P]\!]$ in $\Psi[\![\lambda k x_1 \ldots x_n.P]\!] = \lambda x_1 \ldots x_n.\mathcal{U}[\![M]\!]$. These typos are easy to catch when one tries running the transformations as they appear in the paper; Redex can render typeset figures of models, which authors could use to avoid introducing errors (perhaps such as these) during manual transcription into LaTeX.

References

[1] M. J. Fischer. Lambda-calculus schemata. *LISP and Symbolic Computation*, 6: 259–287, 1993.

[2] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM.

[3] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, pages 288–298, New York, NY, USA, 1992. ACM.