Administrative normal form, continued

Sharing control in direct style

Luke Maurer Paul Downen
Zena M. Ariola
University of Oregon
{pdownen,maurerl,ariola}@cs.uoregon.edu

Simon Peyton Jones
Microsoft Research Cambridge
simonpj@microsoft.com

Abstract

Administrative normal form (ANF) promises to reap the benefits of continuation-passing style (CPS) while retaining the advantages of direct style. We believe ANF falls short of this ideal because it does not provide a satisfactory way to describe shared control flow, as CPS does naturally. We show how CPS itself can guide us in adding a control effect to ANF and taming it to regain purity, guaranteeing an efficient implementation. Interestingly, even though CPS is usually used for compiling call-by-value languages, our technique is independent of the evaluation strategy. In short, we propose that the known advantages of CPS can be achieved by extending ANF or another direct-style representation. Does this mean that CPS is finally out? No: whatever the compiler's chosen representation, we suspect CPS will reveal new insights about program manipulation.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors—Compilers

Keywords Intermediate representations; Compiler optimizations; Control effects

Talk proposal

Continuation-passing style has a storied history both in theory and in the implementation of compilers. It advanced the theory of the call-by-value λ-calculus by providing new tools to study execution behavior, validating new axioms for program equivalence [12]. It has also helped devise and compare powerful new control operators [14]. For compiler implementors, CPS provides a way of representing control flow that reuses well-understood λ-calculus constructs but readily translates to assembly code [1, 13]. Furthermore, it can express many features, such as state and control, without adding new language constructs; in fact, any monadic effect can be expressed using CPS [4].

Yet CPS has sometimes fallen out of favor as an internal representation. It requires a whole-program transformation that leaves the code with a radically different structure. It can increase the size of the terms significantly, and the transformed terms take more technical knowledge to understand. Furthermore, the benefits to code generation may not be appreciable earlier in the pipeline—to the optimizer, having a name for each evaluation context may only provide unnecessary details.

Direct-style representations have strengths of their own as well. Common subexpressions are simple to recognize and eliminate. Also, when the internal representation is merely a distillation of the original program, one can formulate rewrites in terms of the source language. The Glasgow Haskell Compiler (GHC) takes advantage of this by allowing user code to specify rewrite rules for the optimizer to apply, thus imparting domain knowledge about specialized datatypes to a general-purpose simplifier [10].

Given that CPS is used to prove things about direct-style programs, one might hope that reductions in CPS programs could be translated into direct style, thus allowing an optimizer to exploit CPS technology without needing to implement it. A famous 1993 paper by Flanagan et al. [5] showed that this was in fact possible: putting code into their *administrative normal form*, or ANF, promises to perform the same transformations that a CPS compiler would. In this way, rather than a day-to-day tool, CPS can be a source of wisdom to be consulted before returning to work in direct style.

So it may seem that, as a compiler technology, CPS is obsolete. However, a 2007 paper by Kennedy [7] argues otherwise, noting that there are significant drawbacks to ANF and other modern direct-style representations. One major issue has to do with code size, which is a major concern for modern processors; increases in size threaten to flood the cache and wipe out the gains made by whatever improvements an optimizer might make.

To see where code size comes into play, consider this function:

$$\begin{aligned} g\,x = f\ (&\mathbf{case}\,x\,\mathbf{of}\ A \to 1 \\ &\qquad\qquad\ \ B \to 2) \end{aligned} \tag{1}$$

In call-by-value (or if f is strict), we would like to move the application of f into the branches of the **case**. Such a *commuting conversion*—a term originating from proof theory [6]—is sound because whichever branch is taken, its value will eventually be the argument to f anyway. Commuting conversions are useful both to aid code generation and to bring redexes together, enabling more compile-time reductions [9].

There is a risk, however. If we simply move the application inward as is, we produce:

$$\begin{aligned} g\,x = \ &\mathbf{case}\,x\,\mathbf{of}\ A \to f\ 1 \\ &\qquad\qquad\ \ B \to f\ 2 \end{aligned}$$

Where before we had one occurrence of f, we now have two. Here this is benign enough, but f could be an arbitrary expression, or more generally an arbitrary context (e.g., another **case** expression). In general, unrestrained commuting conversions lead to exponential blowup [8].

An obvious solution is to optimize the original function g by introducing a **let**-bound function with the shared code:

$$\begin{aligned} g\,x = \ &\mathbf{let}\,j\ x = f\ x \\ &\mathbf{in}\ \mathbf{case}\,x\,\mathbf{of}\ A \to j\ 1 \\ &\qquad\qquad\qquad\ B \to j\ 2 \end{aligned} \tag{2}$$

Such a function j is usually called a *join point*. A join point represents a place where multiple branches of control flow come together (in other words, where they *join*). In this sense, they are much like the ϕ-nodes in the SSA languages favored in the imperative world [3].

Of course, all we have really done here is to reinvent the continuation (as has happened many times before [11]). Nonetheless, join points are a simple and effective way to manage code size when doing commuting conversions. In addition, they can compile to fast code: since every invocation of a join point is a tail call, they can be represented by simple labeled blocks of code and invoked by jumping. Thus the actual increase in function-call overhead is reduced sharply.

Unfortunately, the story doesn't end there. Once a join point is created, there may be any number of further transformations applied to the program—including more commuting conversions. Suppose we call g as the argument to another function, and g ends up being inlined:

$$a\;x = h\;(g\;x) \qquad \rightarrow \qquad \begin{aligned} a\;x = h\;(&\mathbf{let}\,j\;x = f\;x \\ &\mathbf{in\;case}\,x\,\mathbf{of}\;A \rightarrow j\;1 \\ &\qquad\qquad\quad\; B \rightarrow j\;2) \end{aligned}$$

As before, we want to perform commuting conversions that push the call to h downward:

$$\begin{aligned} a\;x = \;&\mathbf{let}\,j\;x = f\;x \\ &\mathbf{in\;case}\,x\,\mathbf{of}\;A \rightarrow h\;(j\;1) \\ &\qquad\qquad\quad\; B \rightarrow h\;(j\;2) \end{aligned} \tag{3}$$

Sadly, we can no longer compile j as a block, since the calls to it aren't tail calls. It no longer acts as a join point, and so it incurs full function-call overhead.[1] In the end, our method of sharing code paths at low cost was fragile to further rewrites.

One solution is to be more careful doing commuting conversions. This approach was taken by Benton et al. in developing SML.NET [2]. Their compiler (at the time) used a monadic intermediate representation rather than ANF, but ran into an issue with an example very similar to ours. Their solution was to mind the order in which commuting conversions were performed, so as to avoid lifting the same **case** more than once. However, this only avoids the issue in the span of a single normalization pass. We would prefer a method that is robust to other transformations; there may be several different optimization passes, each of which triggers a round of normalization and commuting conversions.

Let us see how CPS behaves in this situation. Here is the CPS translation of g as we first defined it in (1):

$$\begin{aligned} g_{cps}\;(x,k) = \;&(\lambda k'.\,\mathbf{case}\,x\,\mathbf{of}\;A \rightarrow k'\;1 \\ &\qquad\qquad\qquad\quad B \rightarrow k'\;2) \\ &(\lambda v.f\;(v,k)) \end{aligned}$$

In place of the commuting conversion, we have a simple β-reduction. Moreover, if duplication is a concern, we can simply bind the continuation to a name as we might any value argument:

$$\begin{aligned} g_{cps}\;(x,k) = \;&\mathbf{let}\,k'\;v = f\;(v,k) \\ &\mathbf{in\;case}\,x\,\mathbf{of}\;A \rightarrow k'\;1 \\ &\qquad\qquad\quad\; B \rightarrow k'\;2 \end{aligned}$$

Note the similarity to the definition of g with a join point given in (2).

Now, what happens when the optimized g_{cps} is inlined?

$$a_{cps}\;(x,k) = g_{cps}\;(x, \lambda w.h\;(w,k))$$

This time, the call to h is substituted *into the shared continuation*:

$$\begin{aligned} a_{cps}\;(x,k) = \;&\mathbf{let}\,k'\;v = f\;(v, \lambda w.h\;(w,k)) \\ &\mathbf{in\;case}\,x\,\mathbf{of}\;A \rightarrow k'\;1 \\ &\qquad\qquad\quad\; B \rightarrow k'\;2 \end{aligned}$$

Since k' is still only ever tail-called, it can still be compiled as a block. CPS has shown us the way: a commuting conversion should bring the evaluation context into the join point, not its call sites.

As we have seen, CPS compilers can manage code size by sharing continuations. Sharing code among different execution paths is as easy as using a name in multiple places. This convenience is a consequence of the way CPS collapses control flow with data flow: sharing control becomes exactly the same as sharing values. The direct-style λ-calculus does not exhibit this symmetry; once a CPS program has a shared continuation like in g_{cps}, it no longer corresponds to any direct-style program.

Now, let's see if we can put what we've learned into direct style. Suppose we distinguish join points from other bindings using the **label** keyword, and then we invoke join points using the jump keyword. When we perform the first commuting conversion, we end up here:

$$\begin{aligned} g'\;x = \;&\mathbf{label}\,j\;v = f\;v \\ &\mathbf{in\;case}\,x\,\mathbf{of}\;A \rightarrow \mathsf{jump}\,j\;1 \\ &\qquad\qquad\quad\; B \rightarrow \mathsf{jump}\,j\;2 \end{aligned}$$

Again, we inline g' into another function call:

$$\begin{aligned} a'\;x = h\;(&\mathbf{label}\,j\;v = f\;v \\ &\mathbf{in\;case}\,x\,\mathbf{of}\;A \rightarrow \mathsf{jump}\,j\;1 \\ &\qquad\qquad\quad\; B \rightarrow \mathsf{jump}\,j\;2) \end{aligned}$$

At this point, however, we follow CPS's suggestion and perform the second commuting conversion like this:

$$\begin{aligned} a'\;x = \;&\mathbf{label}\,j\;v = h\;(f\;v) \\ &\mathbf{in\;case}\,x\,\mathbf{of}\;A \rightarrow \mathsf{jump}\,j\;1 \\ &\qquad\qquad\quad\; B \rightarrow \mathsf{jump}\,j\;2 \end{aligned}$$

Compare to the optimized a in (3). The application of h now moves *into the join point* j and disappears from j's call sites. If there were a third branch that didn't invoke the join point, say $C \rightarrow 3$, it would take on the new evaluation context as usual, becoming $C \rightarrow h\;3$. All of this follows directly from the way substitution occurs on continuations in CPS.

This technique is very general. Notice that evaluation order was never important to our discussion; all that matters is that the **case** occurs in a strict context (as in g and again in a). We used call-by-value in the examples simply for brevity, but the idea applies to any language, lazy or eager.[2] Of course, as we are proposing adding labels and jumps to a direct-style language, it would appear we now have a control calculus.[3] Interestingly, we tame the control effect with simple scope restrictions on labels, regaining the purity of the language. Thus, we have made join points robust while still being able to compile them to fast code.

So, is this the end for CPS? Have we improved ANF to the point where we can drop CPS from our compilers and never look back? We hesitate to claim so. After all, tomorrow someone may find something *else* we can learn from CPS. Or perhaps there are lessons in CPS with state, or CPS with exceptions, or some other variation. In any case, while CPS may go in and out of fashion as an internal representation, it will long have things to teach theoreticians and practitioners alike.

[1] We might also be concerned with the duplication of h here, but of course that simply calls for another join point. This, however, would not address the problem with j.

[2] In fact, we have been implementing these ideas for compiling Haskell.

[3] Indeed, we have: $\mathit{callcc}\;f = \mathbf{label}\,\mathit{here}\;x = x\,\mathbf{in}\,f\;(\lambda x.\,\mathsf{jump}\,\mathit{here}\;x)$

References

[1] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992. ISBN 0-521-41695-7.

[2] N. Benton, A. Kennedy, S. Lindley, and C. V. Russo. Shrinking reductions in SML.NET. In *Implementation and Application of Functional Languages, 16th International Workshop, IFL 2004, Lübeck, Germany, September 8-10, 2004, Revised Selected Papers*, pages 142–159, 2004. . URL `http://dx.doi.org/10.1007/11431664_9`.

[3] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991. . URL `http://doi.acm.org/10.1145/115372.115320`.

[4] A. Filinski. Representing monads. In *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, pages 446–457, 1994. . URL `http://doi.acm.org/10.1145/174675.178047`.

[5] C. Flanagan, A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247, 1993. . URL `http://doi.acm.org/10.1145/155090.155113`.

[6] J.-Y. Girard, P. Taylor, and Y. Lafont. *Proofs and types*, volume 7. Cambridge University Press Cambridge, 1989.

[7] A. Kennedy. Compiling with continuations, continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming, ICFP 2007, Freiburg, Germany, October 1-3, 2007*, pages 177–190, 2007. . URL `http://doi.acm.org/10.1145/1291151.1291179`.

[8] S. Lindley. *Normalisation by evaluation in the compilation of typed functional programming languages*. PhD thesis, University of Edinburgh, College of Science and Engineering, School of Informatics, 2005.

[9] S. Peyton Jones and A. Santos. A transformation-based optimiser for Haskell. *Science of Computer Programming*, 32(1-3):3–47, Sept. 1998.

[10] S. L. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *2001 Haskell Workshop*. ACM SIGPLAN, September 2001. URL `http://research.microsoft.com/apps/pubs/default.aspx?id=74064`.

[11] J. C. Reynolds. The discoveries of continuations. *Lisp and Symbolic Computation*, 6(3-4):233–248, 1993.

[12] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. In *LISP and Functional Programming*, pages 288–298, 1992. . URL `http://doi.acm.org/10.1145/141471.141563`.

[13] G. L. Steele, Jr. RABBIT: A compiler for SCHEME. Technical Report AITR-474, Massachusetts Institute of Technology, Artificial Intelligence Laboratory, 1978.

[14] H. Thielecke. Comparing control constructs by double-barrelled CPS. *Higher-Order and Symbolic Computation*, 15(2-3):141–160, 2002. . URL `http://dx.doi.org/10.1023/A:1020887011500`.