Local CPS conversion in a direct-style compiler

John Reppy
Bell Labs, Lucent Technologies
jhr@research.bell-labs.com

Abstract

Local CPS conversion is a compiler transformation for improving the code generated for nested loops by a direct-style compiler. The transformation consists of a combination of CPS conversion and light-weight closure conversion, which allows the compiler to merge the environments of nested recursive functions. This merging, in turn, allows the backend to use a single machine-level procedure to implement the nested loops. Preliminary experiments with the Moby compiler show the potential for significant reductions in loop overhead as a result of Local CPS conversion.

1 Introduction

Most compilers for functional languages use a λ-calculus based intermediate representation (IR) for their optimization phases. The λ-calculus is a good match for this purpose because, on the one hand, it models surface-language features like higher-order functions and lexical scoping, while, on the other hand, it can be transformed into a form that is quite close to the machine model.

To make analysis and optimization more tractable, compilers typically restrict the IR to a subset of the λ-calculus. One such subset is the so-called *direct style* (DS) representation, where terms are normalized so that function arguments are always atomic (*i.e.*, variables and constants) and intermediate results are bound to variables.[1] The DS representation makes the data-flow of the program explicit by binding all intermediate values to variables. Another common representation is *continuation-passing style* (CPS), where function applications are further restricted to occur only in tail positions and function returns are represented explicitly as the tail-application of continuation functions [Ste78, KKR+86, App92]. In CPS, both the data-flow and control-flow of the program is made explicit, which makes it well suited to optimizing the program's control structures.

While there has been some debate over the relative merits of these two approaches [FSDF93, DD00], it is fair to say that both have their advantages and we do not discuss their relative merits. In the end, the choice of IR is an engineering decision that must be made for each compiler. A discussion of this choice is beyond the scope of this paper; instead, we focus on the idea of exploiting CPS representation in a DS-based optimizer. This exploitation is possible because the CPS terms are a subset of the DS terms (*i.e.*, CPS $\subset$ DS $\subset$ λ-calculus).

This paper describes a transformation and supporting analysis that exemplifies the idea of exploiting CPS representation in a DS-based optimizer. In the next section, we describe a motivating example. We then describe our transformation and an analysis for detecting when it is applicable in Section 3. This transformation should be useful for any DS-based optimizer. We are implementing this transformation in our compiler for the MOBY programming language [FR99] and we present a preliminary indication of its usefulness in Section 4.[2] We discuss related work in Section 5 and then conclude.

2 The problem

It is well known that loops can be represented as tail-recursive functions and many compilers for functional languages use tail recursion to represent loops in their IR. By treating tail-recursive function calls as "gotos with arguments," a compiler can generate code for a loop that is comparable to that generated by a compiler for an imperative language. But when loops are nested, generating efficient code becomes more difficult. For example, consider the following C-code:

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++)
    f (i, j);
```

This kind of nested loop structure is found in many algorithms (*e.g.*, matrix multiplication). Translating this code to a sugared DS representation, with the `for`-loops replaced by recursion, results in the code given in Figure 1.[3] While the two loop functions, `lp_i` and `lp_j`, are tail-recursive, the call "`lp_j 0`" from the outer loop (`lp_i`) to the inner loop is not tail recursive. If the compiler directly translates the DS representation to machine code, the two loops will occupy separate procedures with separate stack frames. This structure inhibits loop optimizations, register allocation, and scheduling, as well as adding call/return overhead to the outer loop. On the other hand, the CPS representation of this example, given in Figure 2, makes explicit the connection between the return from `lp_j` and the next iteration of `lp_i`. A simple control-flow analysis will show that the return continuation of `lp_j` is always the known function `k5`, which enables compiling the nested loops into a single machine procedure.

[1] There are a number of different direct-style representations: *e.g.*, Flanagan *et al's A-form* [FSDF93], the TIL compiler's *B-form* [TMC+96], and the RML compiler's *SIL* [OT98].

[2] MOBY is a higher-order typed language that combines support for functional, object-oriented, and concurrent programming. See `www.cs.bell-labs.com/~jhr/moby` for more information.

[3] In this paper we use SML-like syntax as a sugared form of DS representation.

```
fun applyf (f, n) = let
      fun lp_i i = if (i < n)
            then let
              fun lp_j j = if (j < n)
                  then (f(i, j); lp_j(j+1))
                  else ()
              in
                lp_j 0; lp_i(i+1)
              end
            else ()
      in
        lp_i 0
      end
```

Figure 1: A nested loop using tail-recursion

```
fun applyf (f, n, k1) = let
      fun lp_i (i, k2) = if (i < n)
            then let
              fun lp_j (j, k3) = if (j < n)
                  then let
                    fun k4 () = lp_j(i+1, k3)
                    in
                      f(i, j, k4)
                    end
                  else k3()
              fun k5 () = lp_i(i+1, k2)
              in
                lp_j (0, k5)
              end
            else k2()
      in
        lp_i (0, k1)
      end
```

Figure 2: The CPS converted `applyF` example

This example suggests that by making the return continuation of `lp_j` explicit, we can replace the call/return of `lp_j` with direct jumps.

3 The solution

The MOBY compiler performs standard optimizations (*e.g.*, contraction, useless-variable elimination, and CSE) using a DS representation we call BOL. Loops are represented using tail recursion in BOL. After optimization, the compiler performs the *closure* phase, which is responsible for converting the nested functions into a collection of top-level functions with no free variables. Following the closure phase is the *frame* phase, which determines which functions can share the same stack frame; a group of functions that share the same frame is called a *cluster*. Our goal is to have nested loops, like the one in Figure 1, translate into a single cluster (as they would in an imperative language like C). Doing so has many performance advantages. It enables better loop optimizations, register allocation, and scheduling. It also eliminates the overhead of creating a closure for the inner loop, the call to the inner loop, and the heap-limit check on return from the inner loop.

The technique we use to achieve this goal is us a *local CPS* (LCPS) conversion to convert the non-tail calls to the inner loop into tail calls. Once the LCPS transformation has been applied, the frame phase is able to group the functions that comprise the nested loop into the same cluster.

$$
\begin{array}{rcll}
e & ::= & l : t & \text{labeled term} \\
t & ::= & x & \text{variable} \\
 & | & \textbf{fun}\ f\,(\vec{x}) = e_1\ \textbf{in}\ e_2 & \text{function binding} \\
 & | & \textbf{let}\ x = e_1\ \textbf{in}\ e_2 & \text{let binding} \\
 & | & \textbf{if}\ x\ \textbf{then}\ e_1\ \textbf{else}\ e_2 & \text{conditional} \\
 & | & f\,(\vec{x}) & \text{application}
\end{array}
$$

Figure 3: A simple direct-style IR

To determine where it is useful to apply the transformation, we need some form of control-flow analysis. The property that we are interested in is when a known function has the same return continuation at all of its call-sites. The MOBY compiler uses a simple syntactic analysis to determine this property. For each function defined in the module being compiled, we conservatively estimate the set of return continuations for the function. If the estimated set is a singleton set, then we apply the transformation.

3.1 Analysis

The analysis computes an approximation of return continuations of each known function, so a standard control-flow analysis is applicable [NNH99]. In this section, we describe a very simple linear-time analysis. This analysis uses a simple notion of *escaping* function — if a function name is mentioned in a non-application rôle, it is regarded as escaping and we define its return continuation to be $\top$.[4]

To describe the analysis, we use the simple DS IR given in Figure 3. As usual, we assume bound variables are unique so we do not have to worry about unintended name capture when transforming code. In this IR, expressions are uniquely labeled terms. We use the labels to represent abstract continuations in the analysis.

Let LABEL be the set of term labels. Then we define an abstract domain $\text{RCONT} = \text{LABEL} \cup \{\bot, \top\}$. We use ρ to denote elements of RCONT. Intuitively, one can think of RCONT as a squashed powerset domain, with $\bot$ for the empty set, l for the singleton set $\{l\}$, and $\top$ for everything else. We define the partial order $\sqsubseteq$ on RCONT, with $\bot \sqsubseteq l \sqsubseteq \top$ for any $l \in \text{LABEL}$, and we define $\rho_1 \sqcup \rho_2$ to be the least upper bound of ρ_1 and ρ_2 under $\sqsubseteq$.

Given an expression and its abstract continuation, the analysis computes a map Γ from variables (*i.e.*, function names) to abstract continuations.

$$\Gamma \in \text{RENV} = \text{VAR} \xrightarrow{\text{fin}} \text{RCONT}$$

We extend Γ to a total function when applying it to a variable by defining $\Gamma(x) = \bot$ for $x \notin \text{dom}(\Gamma)$. We define the *join* of Γ_1 and Γ_2 by

$$\Gamma_1 \uplus \Gamma_2 = \{x \mapsto \Gamma_1(x) \sqcup \Gamma_2(x) \mid x \in \text{dom}(\Gamma_1) \cup \text{dom}(\Gamma_2)\}$$

The analysis itself has the following type:

$$\mathcal{R} : \text{EXP} \to \text{RCONT} \to \text{RENV}$$

With these definitions, we can describe the analysis, which is presented in Figure 4. We map unknown and escaping functions to $\top$, as can be seen in Rules 1, 2, and 5. Rule 2 shows how we analyse function definitions — first we analyse the uses of f in its scope and then we use the result of that analysis as the return

[4] This definition is the one used by Appel [App92] in his CPS-based framework.

$$
\begin{aligned}
\mathcal{R}[\![l : x]\!]\rho &= \{x \mapsto \top\} && (1)\\
\mathcal{R}[\![l : \texttt{fun}\ f\,(\vec{x}) = e_1\ \texttt{in}\ e_2]\!]\rho &= \mathcal{R}[\![e_1]\!]\rho' \uplus \Gamma \uplus \{\vec{x} \mapsto \top\} && (2)\\
&\quad \text{where } \Gamma = \mathcal{R}[\![e_2]\!]\rho \text{ and } \rho' = \Gamma(f).\\
\mathcal{R}[\![l : \texttt{let}\ x = e_1\ \texttt{in}\ l' : t_2]\!]\rho &= \mathcal{R}[\![e_1]\!]l' \uplus \mathcal{R}[\![l' : t_2]\!]\rho \uplus \{x \mapsto \top\} && (3)\\
\mathcal{R}[\![l : \texttt{if}\ x\ \texttt{then}\ e_1\ \texttt{else}\ e_2]\!]\rho &= \mathcal{R}[\![e_1]\!]\rho \uplus \mathcal{R}[\![e_2]\!]\rho && (4)\\
\mathcal{R}[\![l : f\,(\vec{x})]\!]\rho &= \{f \mapsto \rho\} \uplus \{\vec{x} \mapsto \top\} && (5)
\end{aligned}
$$

Figure 4: The analysis

continuation for the body of f. For **let** bindings, the body of the **let** is the continuation of the binding. The result of the analysis is the join of the sub-analyses. When analysing a function application (Rule 5), we map the applied function to the application's abstract continuation and treat the arguments as escaping. To analyse a complete program, we use $\top$ as the return continuation and define $\text{ANAL}(e) = \mathcal{R}[\![e]\!]\top$.

This analysis can be extended to handle mutually recursive functions by computing a fixed-point at function bindings. Since such a brute-force approach may prove expensive in practice, a better solution may be to first compute the approximate call-graph and then use the call-graph to guide the analysis.

3.2 The LCPS transformation

If the analysis has determined that all call sites of a function `f` have the same return continuation (*i.e.*, $\text{RENV}(\texttt{f}) = l$), then we apply the LCPS transformation to `f`. Transforming `f` has two parts: we must reify the continuation of `f`, creating an *explicit* continuation function $k_{\texttt{f}}$, and we must introduce calls to $k_{\texttt{f}}$ at the return sites of `f`. For example, consider the following code fragment:

```
fun f () = ... in ...
  fun g () = ... let y = f() in e
```

Assuming that f is eligible for the LCPS transformation, this fragment is converted to

```
fun f (k) = (... k()) in ...
  fun g () = ... fun kf (y) = e in f(kf)
```

Here we have made the return continuation of `f` explicit by modifying `f` to take its continuation as an argument (k), which it calls at its return sites. We have also split the body of `g` to create the explicit representation of `f`'s return continuation ($k_{\texttt{f}}$) and have modified the non-tail call site of `f` to pass $k_{\texttt{f}}$ to `f`.

To understand how the LCPS transformation works, it is instructive to examine the global CPS conversion. Figure 5 gives the transformation for the simple direct-style IR of Figure 3 (ignoring the labels). For the LCPS transformation, we only want to apply the CPS conversion under certain conditions. Assuming that Γ is the result of analysing the program, let the set of eligible function be defined as $\mathcal{E} = \{f \mid \Gamma(f) \in \text{LABEL}\}$. We then specialize the rules of Figure 5 as follows:

Rule 6: When the expression x is in the tail position of a function $f \in \mathcal{E}$, then we apply the CPS conversion (*i.e.*, we transform the implicit return into an explicit application of the tail continuation k).

Rule 7: When $f \in \mathcal{E}$ we apply the CPS conversion.

Rule 8: When the expression e_1 contains an application of a function $f \in \mathcal{E}$, we apply the CPS transformation.

Rule 9: Since this rule does not transform the expression, there is no modification. Note that we do not have to worry about code duplication, since k is a variable and not a λ-term.

Rule 10: If the function $f \in \mathcal{E}$, then we apply the CPS conversion. In this situation, k will either be an explicit return continuation (introduced by Rule 8) or a return continuation parameter (introduced by Rule 7).

It is interesting to examine what happens when the call-site of `f` is buried in the branch of a conditional. For example, consider the following fragment:

```
let x = if y then (let w = f() in e1) else e2
  in e3
```

Applying LCPS to `f` results in the following code:

```
fun kif (x) = e3 in
  if y
    then fun kf (w) = C[[e1]]kif in f(kf)
    else C[[e2]]kif
```

Here we have introduced two continuation functions: $k_{\texttt{if}}$ for the join-point following the **if** and $k_{\texttt{f}}$ for the return continuation of `f`.

Up to now, we have been passing the return continuation $k_{\texttt{f}}$ to `f` as an additional argument (as is standard in CPS conversion), but since our analysis has already told us that `f` has exactly one return continuation, we should specialize the return sites of `f` to call $k_{\texttt{f}}$ directly. To do so requires lifting the definition of $k_{\texttt{f}}$ up to the binding site of `f`.[5] The difficulty with this lifting is that the return continuation and its functions have different free variables, so we need to *close* the function over those variables that will be out of scope at the destination of the function. While a closure object would be sufficient for this purpose, we chose to augment our transformation with a simple form of *light-weight closure conversion* [SW97], which turns the free variables into function parameters. Having these variables as parameters in the final DS representation means that they will get mapped to machine registers. To illustrate closure conversion, consider the following fragment:

```
fun f () = ... z in
...
  let y = ... in
  ...
    let x = f() in
    let w = x + y in
      w
```

In this code, the return continuation of the call to `f` has `y` as a free variable, so we add `y` to the parameters of `f` and the transformed `f` passes `y` to its return continuation. The result of the transformation is:

[5] We also need to extend the IR to support mutually recursive bindings.

$$\mathcal{C}[\![x]\!]k = k(x) \quad (6)$$

$$\mathcal{C}[\![\textbf{fun}\ f(\vec{x}) = e_1\ \textbf{in}\ e_2]\!]k = \textbf{fun}\ f(k', \vec{x}) = \mathcal{C}[\![e_1]\!]k'\ \textbf{in}\ \mathcal{C}[\![e_2]\!]k \quad \text{where } k' \text{ is fresh} \quad (7)$$

$$\mathcal{C}[\![\textbf{let}\ x = e_1\ \textbf{in}\ e_2]\!]k = \textbf{fun}\ k'(x) = \mathcal{C}[\![e_2]\!]k\ \textbf{in}\ \mathcal{C}[\![e_1]\!]k' \quad \text{where } k' \text{ is fresh} \quad (8)$$

$$\mathcal{C}[\![\textbf{if}\ x\ \textbf{then}\ e_1\ \textbf{else}\ e_2]\!]k = \textbf{if}\ x\ \textbf{then}\ \mathcal{C}[\![e_1]\!]k\ \textbf{else}\ \mathcal{C}[\![e_2]\!]k \quad (9)$$

$$\mathcal{C}[\![f(\vec{x})]\!]k = f(k, \vec{x}) \quad (10)$$

Figure 5: A global CPS conversion

```
fun applyf (f, n) = let
      fun lp_i (i, f, n) = if (i < n)
            then lp_j (0, i, f, n)
            else ()
      and lp_j (j, i, f, n) =
          if (j < n)
            then (
              f(i, j);
              lp_j(j+1, i, f, n))
            else k (i, f, n)
      and k (i, f, n) =
          lp_i(i+1, f, n)
      in
        lp_i (0, f, n)
      end
```

Figure 6: The `applyF` function after the LCPS transformation

```
fun f (y) = ... kf(z, y)
and kf (x, y) = let w = x + y in w
...
  let y = ... in
  ...
    f(y)
```

We define `f` and k_f in the same binding because, in general, they may be mutually recursive. In general, the LCPS transformation migrates the converted function f and its explicit continuation to the binding site of f's *non-tail* caller. By doing so, we guarantee that all these functions will be compiled into a single machine procedure.

Revisiting our original motivating example from Figure 1, the result of the analysis will identify `lp_j` as a candidate for the LCPS transformation (*i.e.*, all of its call sites have the same return continuation). The code resulting from applying the transformation to `lp_j` is given in Figure 6. Notice that light-weight closure conversion has been applied to all of the functions in the body of `applyF`. When translated to the target machine code, the functions `applyF`, `lp_i`, `lp_j`, and `k` will all be in the same cluster and share the same stack frame.

4 Experience

We have written a prototype implementation of the analysis and LCPS transformation for the language of Figure 3. Our implementation is currently organized into the analysis plus four transformation passes. The first pass marks those **`let`** bindings for which Rule 8 applies and the second pass performs the actual CPS transformation guided by the marks.[6] The second pass also determines which functions should share the same binding site. The third pass then computes the free variables of these functions and the fourth pass migrates the function definitions and performs the light-weight closure conversion.

We are in the process of implementing the LCPS transformation in the MOBY compiler. For the most part, this is straightforward adaptation of our prototype, although we need a more sophisticated analysis to handle mutually recursive functions. We are also integrating the third and fourth passes of the transformation with the existing closure phase. This integration has the added benefit that we can use light-weight closure conversion for functions that represent local control-flow (*e.g.*, loops). By moving variables from closures into parameters, we expose them to the register allocator and reduce heap allocation,

To judge the effectiveness of the transformation, we applied it as a source-language transformation to the MOBY version of the `applyF` function (essentially, we compared the MOBY versions of Figure 1 and Figure 6). Running the `applyF` program on the null function with `n` set to 10000, the LCPS transformation results in a 25% reduction in execution time (2.34s vs. 3.11s on a 733MHz PIII). While the performance improvements on real workloads is yet to be determined, these preliminary measurements strongly suggest that the LCPS transformation is a useful tool in a DS optimizer.

5 Related work

Most of the literature about compiler optimizations for strict functional languages uses CPS as a representation. Tarditi's thesis [Tar96] is probably the most detailed description of a DS-based optimizer for strict functional languages, but he does not collapse nested loops. We are not aware of any direct-style compiler that implements the LCPS transformation (the OCAML [Ler00], TIL [TMC+96, Tar96], and RML [OT98] compilers do not). The OCAML system provides **`for`**-loops over integer intervals as a language feature. These loops are preserved in the IR and result in code that is similar to that produced by a C compiler [Ler97], but the OCAML compiler (Version 3.00) does not flatten nested loops when they are expressed using recursion.

Kelsey describes a technique for combining functions in a CPS-based framework [Kel95]. In his framework, he annotates λ abstractions with either *proc*, *cont*, or *jump*, where jump is used to mark control transfers that occur within the same machine procedure. He describes an analysis and transformation for converting a λ_{proc} into a λ_{jump}. The MOBY compiler's frame phase (mentioned in Section 3) performs a similar analysis and transformation when grouping functions into clusters. Since the BOL IR is direct-style,

[6] We use two passes for this part of the transformation because it greatly simplifies the bookkeeping.

we rely on the LCPS transformation to enable the clustering of nested loops in the frame phase.

The MLton compiler is a CPS-based compiler for Standard ML, which uses a transformation called *contification* to group functions into the same machine procedure. This transformation very similar to Kelsey's approach, but it has not been described in the literature.

Kim, Yi, and Danvy have used *selective CPS transformation* as a technique for replacing SML's exception raising and handling mechanisms with continuation operations [KYD98]. Unlike LCPS, their transformation does not move code or do closure conversion. Selective CPS transformation is another example of using the CPS representation in a direct-style compiler (although their experiments were done using SML/NJ for their backend, which is a CPS-based compiler).

6 Conclusion

We have presented a local CPS transformation that can be used in a direct-style compiler to improve the performance of nested loops. Preliminary measurements show a 25% reduction in loop overhead for a simple nested loop. While we have only presented fairly simple examples, the LCPS transformation can handle complicated looping structures, such as multiple inner loops and loops expressed as mutually recursive functions (*e.g.*, as you would get when encoding state machines). The LCPS transformation is one example of exploiting the advantages of CPS in a direct-style compiler; we plan to explore other opportunities for exploiting CPS in our compiler.

Acknowledgements

Kathleen Fisher, Lal George, and Jon Riecke provided useful comments on drafts of this paper. Stephen Weeks provided a detailed description of the MLton contification analysis and transformation.

References

[App92] Appel, A. W. *Compiling with Continuations*. Cambridge University Press, New York, N.Y., 1992.

[DD00] Damian, D. and O. Danvy. Syntactic accidents in program analysis: On the impact of the CPS transformation. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, September 2000, pp. 209–220.

[FR99] Fisher, K. and J. Reppy. The design of a class mechanism for Moby. In *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation*, May 1999, pp. 37–49.

[FSDF93] Flanagan, C., A. Sabry, B. F. Duba, and M. Felleisen. The essence of compiling with continuations. In *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation*, June 1993, pp. 237–247.

[Kel95] Kelsey, R. A. A correspondence between continuation passing style and static single assignment form. In *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations*, January 1995, pp. 13–22.

[KKR+86] Kranz, D., R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. Orbit: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*, July 1986, pp. 219–233.

[KYD98] Kim, J., K. Yi, and O. Danvy. Assessing the overhead of ML exceptions by selective CPS transformation. In *Proceedings of the 1998 ACM SIGPLAN Workshop on ML*, September 1998, pp. 103–114.

[Ler97] Leroy, X. The effectiveness of type-based unboxing. In *Workshop Types in Compilation '97*. Technical report BCCS-97-03, Boston College, Computer Science Department, June 1997.

[Ler00] Leroy, X. *The Objective Caml System (release 3.00)*, April 2000. Available from `http://caml.inria.fr`.

[NNH99] Nielson, F., H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag, New York, NY, 1999.

[OT98] Oliva, D. P. and A. P. Tolmach. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, **8**(4), July 1998, pp. 367–412.

[Ste78] Steele Jr., G. L. Rabbit: A compiler for Scheme. Master's dissertation, MIT, May 1978.

[SW97] Steckler, P. A. and M. Wand. Lightweight closure conversion. *ACM Transactions on Programming Languages and Systems*, **19**(1), January 1997, pp. 48–86.

[Tar96] Tarditi, D. *Design and implementation of code optimizations for a type-directed compiler for Standard ML*. Ph.D. dissertation, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, December 1996. Available as Technical Report CMU-CS-97-108.

[TMC+96] Tarditi, D., G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed compiler optimizing compiler for ML. In *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation*, May 1996, pp. 181–192.