Optimizing Nested Loops Using Local CPS Conversion

John Reppy (jhr@research.bell-labs.com)
Bell Labs, Lucent Technologies
600 Mountain Ave.
Murray Hill, NJ 07901
U.S.A.

Abstract. Local CPS conversion is a compiler transformation for improving the code generated for nested loops by a direct-style compiler that uses recursive functions to represent loops. The transformation selectively applies CPS conversion at non-tail call sites, which allows the compiler to use a single machine procedure and stack frame for both the caller and callee. In this paper, we describe LCPS conversion, as well as a supporting analysis. We have implemented Local CPS conversion in the MOBY compiler and describe our implementation. In addition to improving the performance of loops, Local CPS conversion is also used to aid the optimization of non-local control flow by the MOBY compiler. We present results from preliminary experiments with our compiler that show significant reductions in loop overhead as a result of Local CPS conversion.
An earlier version of this paper was presented at the Third ACM SIGPLAN Workshop on Continuations [25].

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 such as higher-order functions and lexical scoping, while, on the other hand, it can be transformed into a first-order 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 syntactic values (*i.e.*, variables and constants) and all intermediate results are bound to variables.[1] By giving names to all intermediate results, the DS representation makes the data flow of the program explicit (*i.e.*, the left-hand-side binding occurrences of variables correspond to nodes and the right-hand-side use occurrences correspond to edges in a data-flow graph), which eases the implementation of analysis and optimization algorithms. Another common IR in functional-language compilers is *Continuation-Passing Style* (CPS), where function applications are further restricted to occur only in tail positions and function returns are represented

[1] There are a number of different direct-style representations: *e.g.*, Flanagan *et al.*'s *A-form* [8], the TIL compiler's *B-form* [34], and the RML compiler's *SIL* [23].

explicitly as the tail-application of continuation functions [1, 32, 17]. As with DS representations, the names given to intermediate results in a CPS representation make the data flow explicit. Furthermore, the explicit return continuations make the control flow explicit (*i.e.*, calls to continuations correspond to edges and continuation functions correspond to nodes in a control-flow graph), which makes CPS well suited to optimizing the program's control structures. A third class of IRs are the *monadic normal forms*, such as Benton *et al.*'s MIL [3], which are based on the ideas of Moggi's computational λ-calculus [20]. Like DS and CPS, this class of IRs make a syntactic distinction between values and computations; the main difference is that they have a much richer notion of value. The results of this paper should apply to this class of IRs without much change.

The choice between basing a compiler on a DS or CPS IR is one that many feel strongly about [1, 4, 8, 26, 29, 33]. A DS representation more closely follows the original program structure than a CPS representation and is more compact, since it does not explicitly represent return continuations. CPS representations, on the other hand, provide a more natural framework for control-flow optimizations. In the end, the choice of IR is an engineering decision that must be made for each compiler and there are many examples of quality compilers using both techniques (some of these are described in Section 7).

Compilers for functional languages must deal with mapping nested and higher-order functions down to machine code; λ-calculus based IRs provide an effective notation for this translation. Typically, such compilers perform *closure conversion* to convert a higher-order IR to a first-order IR [1, 27, 34]; such first-order IRs can be either DS or CPS. An important property of these first-order IRs is that tail calls to known functions can be mapped to gotos [31]. Essentially, in these IRs, λ abstractions are DAGs of basic blocks and control-flow edges between these DAGs are represented as tail calls to known functions.[2]

This paper describes a transformation, called *Local CPS* (LCPS) conversion, and a supporting analysis that allows a DS-based compiler to increase the number of tail calls to known functions by converting non-tail calls into tail calls. This transformation is especially effective for improving the performance of programs with nested loop structure. LCPS conversion is based on the idea of *locally* applying CPS conversion when it is useful to have explicit return continuations. The transformation is local in the sense that a given application of the transformation only CPS converts the non-tail call's context and is limited to the body of the function containing the non-tail call. This technique is possible because the CPS terms are a subset of the DS terms

[2] In some normal forms, such as CPS, λ abstractions are trees of basic blocks (called *extended basic blocks* in the compiler literature).

(*i.e.*, CPS $\subset$ DS $\subset$ λ-calculus). LCPS conversion should be useful for most DS-based optimizers.

We have implemented LCPS conversion in the MOBY compiler [6].[3] In the next section, we describe the phases and representations of our compiler that provide the setting for this paper. In Section 3, we describe a motivating example. We then present a formal description of LCPS conversion and an analysis for detecting when it is applicable in Section 4 using a simple DS IR. In Section 5 describe how LCPS is implemented in the MOBY compiler and give another example of its use. We present some preliminary performance measurements of our implementation in Section 6. Related work is discussed in Section 7 followed by our conclusions.

2. The Moby compiler

Before describing either the problem addressed by this paper, or the solution, we need to provide some context. The optimizer and code generation phases of the MOBY compiler use two different intermediate representations. The first, called BOL,[4] is a higher-order direct-style representation on which we perform traditional λ-calculus style transformations (*e.g.*, contraction [2], useless-variable elimination, and CSE [1]). Loops are represented using tail recursion in BOL. While BOL is a normalized IR that binds names to all intermediate results, that property is not important to the issues of this paper, so we use a more standard SML-style syntax for the examples. BOL has the standard call-by-value λ-calculus semantics. After optimization, the MOBY compiler performs *cluster conversion* to convert the higher-order BOL representation to a first-order representation with explicit closures called *BOL clusters*. After further simplification and the introduction of heap-limit checks, the cluster representation is translated to assembly code using the MLRISC framework [11, 10]. In the remainder of this section, we describe the details of the BOL cluster representation and give an overview of the cluster conversion phase.

2.1. BOL CLUSTERS

A BOL cluster is a collection of *fragments*, which are functions that contain no nested function definitions. A fragment is essentially a DAG of basic blocks, albeit expressed in a λ-calculus style notation. At runtime, these fragments share the same stack frame; intuitively, a cluster is a representation of

[3] 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.

[4] The name "BOL" does not stand for anything; rather, it is the lexical average of the acronyms "ANF" and "CPS."

a procedure's control-flow graph. Each cluster has a distinguished fragment, called the *entry fragment*, that is the target of any external or non-tail call of the cluster. The other fragments in the cluster are only called by internal tail calls.

More formally, let the *known* fragments of the program be

$$\mathbf{Known} = \{f \mid \text{all of } f\text{'s call sites are known}\}$$

Then a cluster is defined as follows:

DEFINITION 1. *A cluster is a pair $\langle f, \mathcal{F} \rangle$ of a fragment f, called the* entry *fragment, and a set of* internal *fragments $\mathcal{F}$, which satisfy the following properties:*

1. $f \notin \mathcal{F}$,
2. $\mathcal{F} \subseteq \mathbf{Known}$ *(*i.e.*, all call sites of internal fragments are known),*
3. *for any $g \in \mathcal{F}$ and fragment h, if h has a tail call to g then $h \in \{f\} \cup \mathcal{F}$, and*
4. *for any $g \in \mathcal{F}$, $\nexists h$ such that h has a non-tail call to g.*

where a fragment *is a BOL function that contains no nested function bindings. Furthermore, any free variable in a fragment is either the name of a fragment in the same cluster, the name of an entry fragment of some other cluster in the same compilation unit, or the name of a statically allocated variable imported from some other compilation unit.*

Each BOL cluster is mapped to a single machine procedure by the code generator; *i.e.*, its fragments share the same stack frame at run-time. Control transfers inside the cluster are implemented as jumps and local variables, including internal fragment parameters, are mapped to MLRISC pseudo-registers. Maximizing the size of clusters provides more opportunities to the MLRISC register allocator to keep variables in registers and reduce memory traffic, which is crucial to good performance.

2.2. CLUSTER CONVERSION

There are two important aspects to cluster conversion: *frame conversion*, which groups fragments into clusters, and *closure conversion*, which determines the explicit representation of function environments [1]. We perform these conversions without actually transforming the program. Instead, we compute all of the information needed to transform the program in auxiliary

data structures. Once this information has been computed we use it to transform the higher-order BOL representation into the first-order BOL cluster representation.[5]

The compiler performs cluster conversion by first constructing a *known call graph* (*i.e.*, an approximate call graph where edges correspond to known function calls) and using that graph to determine a mapping from functions in the higher-order BOL representation to clusters and fragments. The algorithm that maps functions to clusters is similar to Kelsey's algorithm for merging functions [13, 14]; it is designed to maximize the size of clusters. We then compute the free variables of each cluster and determine its closure representation. The free variables are computed on a per-cluster basis (*i.e.*, all of the fragments in a cluster share the same closure) using the original higher-order representation in the standard way [12]. If a variable is free in a fragment, but is bound elsewhere in the same cluster, then we arrange for it to be passed as an explicit parameter to the fragment using a technique that is reminiscent of *light-weight closure conversion* [30]. For example, consider the following BOL representation of a function that applies `f` to the elements of an array `a`:

```
fun applyToArray (f, a) = let
      val n = length a
      fun lp i = if (i < n)
            then let
              val x = sub(a, i)
              in f x; lp(i+1) end
            else ()
      in
        lp 0
      end
```

Cluster conversion maps this code to single cluster consisting of two fragments: the entry fragment `applyToArray` and an internal fragment `lp`. The `lp` fragment has `f`, `a`, and `n` as free variables; light-weight closure conversion maps these to parameters. This results in the following BOL cluster representation:

[5] Our first implementation split these into two separate passes, with closure conversion first producing singleton clusters that were then merged into larger clusters by frame conversion [24].

```
fun applyToArray (f, a) = let
      val n = length a
      in
        lp (0, f, a, n)
      end
and lp (i, f', a', n') = if (i < n')
      then let
        val x = sub(a', i)
        in
          f' x;
          lp(i+1, f', a', n')
        end
      else ()
```

When writing a cluster, we write the entry fragment (`applyToArray` in this case) first. When we translate this representation to MLRISC, the variables `f'`, `a'`, and `n'` are mapped to pseudo registers. The MLRISC register allocator then assigns these to machine registers or local stack variables, depending on the available registers. To simplify the examples in this paper, we do not include the conversion of functions to their explicit closure representation.

3. The problem

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, which is typical of the nested loop structure found in many algorithms (*e.g.*, matrix multiplication):

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

In a DS representation, the **`for`**-loops are represented by tail-recursive functions. Figure 1 gives this example using our SML-like syntax as a sugared form of DS representation. Note that 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. This non-tail call forces the frame phase to map `lp_i` and `lp_j` to different clusters (by Property 4 of Definition 1). When the resulting clusters are translated to assembly code, the two loops will occupy separate procedures with separate stack frames. Splitting the loops across two different clusters inhibits loop optimizations, register allocation, and scheduling, as well as adding call/return overhead to the outer loop.

```
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;  (* not tail recursive *)
                lp_i(i+1)
              end
            else ()
      in
        lp_i 0
      end
```

Figure 1. A nested loop expressed as nested tail-recursive functions

```
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

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.

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.

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

Figure 3. A simple direct-style IR

4. The solution

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 instruction 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.

To achieve the goal of a single merged cluster, we need to address the problem of the non-tail call from `lp_i` to `lp_j`. Our technique is to use *Local CPS* (LCPS) conversion to convert the non-tail call and return into a pair of tail calls. Once LCPS conversion has been applied, it is possible to group the functions that comprise the nested loop into the same cluster.

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. 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.

In the remainder of this section, we give a formal description of LCPS and a simple syntactic analysis of when LCPS can be applied. We also describe some of the issues that a compiler must address in an implementation of LCPS. We use the simple DS IR given in Figure 3 for this formal presentation (Section 5 describes the actual implementation in our compiler). 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.

4.1. Analysis

The analysis computes an approximation of return continuations of each known function, so a standard control-flow analysis is applicable [22]. In this section, we describe a one-pass, linear-time, analysis that uses the term labels as an ab-

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

Figure 4. The analysis

straction of return continuations (we call these *abstract continuations*). 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 $\top$.[6]

Let LABEL be the set of term labels. Then we define an abstract domain RCONT $=$ LABEL $\cup \{\bot, \top\}$ of return continuations. 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 $l \in$ 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 by defining $\Gamma(x) = \bot$ for $x \notin \mathrm{dom}(\Gamma)$. We define the *join* of Γ_1 and Γ_2 by

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

The analysis itself has the following type:

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

and we write $\mathcal{R}[\![e]\!]\rho$ for the analysis of an expression e with the abstract return continuation ρ. 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 continuation for the body of f. For **`let`** bindings, the body of the **`let`** is the continuation of the binding. The

[6] This definition is the one used by Appel [1] in his CPS-based framework.

result of analysing a conditional (Rule 4) is the join of the sub-analyses of its arms. 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 $\text{ANALYSE}(e) = \mathcal{R}[\![e]\!]\top$. The termination of this analysis follows from the fact that it is defined inductively over the structure of the program. We do not need fixed point computations, because the simple language we are using does not have mutually recursive functions (Section 5.1 discusses how the implementation handles mutual recursion).

4.2. LCPS CONVERSION

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

$$\begin{array}{l}\textbf{fun}\ f\ ()\ =\ \ldots\ w\ \textbf{in}\ \ldots\\ \quad\textbf{fun}\ g\ ()\ =\ \ldots\ \textbf{let}\ y\ =\ f()\ \textbf{in}\ \mathit{exp}\end{array}$$

where w is returned as the result of f. Assuming that f is eligible for LCPS conversion, this fragment is converted to

$$\begin{array}{l}\textbf{fun}\ f\ (k)\ =\ (\ldots\ k(w))\ \textbf{in}\ \ldots\\ \quad\textbf{fun}\ g\ ()\ =\ \ldots\ \textbf{fun}\ k_f\ (y)\ =\ \mathit{exp}\ \textbf{in}\ f(k_f)\end{array}$$

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_f) and have modified the non-tail call site of f to pass k_f to f.

To understand how LCPS conversion works, it is instructive to examine the global CPS conversion. Figure 5 gives the transformation for the simple IR of Figure 3 (ignoring the labels). For LCPS conversion, 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 functions be defined as $\mathcal{E} = \{f \mid \Gamma(f) \in \text{LABEL}\}$. We can describe LCPS conversion, which is given in Figure 6, as a modification of CPS conversion from Figure 5.

Rule 6: When the expression x is in the tail position of an eligible function $f \in \mathcal{E}$, then we apply the CPS conversion, as is shown in Rule 12. We use the symbol $\Diamond$ to signal the situation where CPS conversion should not be performed (Rule 11).

Rule 7: We apply the CPS conversion when $f \in \mathcal{E}$ (Rule 13).

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

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

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

$$\mathcal{C}[\![\mathbf{if}\ x\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!]k = \mathbf{if}\ x\ \mathbf{then}\ \mathcal{C}[\![e_1]\!]k\ \mathbf{else}\ \mathcal{C}[\![e_2]\!]k \tag{9}$$

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

Figure 5. A global CPS conversion

Rule 8: When the expression e_1 contains an application of some eligible function $f \in \mathcal{E}$ (signified by the $\mathcal{S}$ predicate being true) we apply the CPS transformation (Rule 14).

Rule 9: Since this rule does not transform the expression, we only need to apply the transformation to the arms of the conditional. Note that code duplication is not a problem, since k is a variable and not a λ-term.

Rule 10: If the continuation is $\Diamond$, which implies that $f \notin \mathcal{E}$, then the application is unchanged (Rule 16). Otherwise, if the function f is eligible (*i.e.*, $f \in \mathcal{E}$), then we apply CPS conversion (Rule 17). In this situation, k will either be an explicit return continuation (introduced by Rule 14) or a return continuation parameter (introduced by Rule 13). If f is not eligible, then we apply the continuation k to its result.

To transform a program e, we compute $\mathcal{L}[\![e]\!]\Diamond$.

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

$$\begin{array}{l}\mathbf{let}\ x = \mathbf{if}\ y\ \mathbf{then}\ (\mathbf{let}\ w = f()\ \mathbf{in}\ exp_1)\ \mathbf{else}\ exp_2 \\ \quad \mathbf{in}\ exp_3\end{array}$$

Applying LCPS to f results in the following code:

$$\begin{array}{l}\mathbf{fun}\ k_{if}\ (x) = exp_3\ \mathbf{in} \\ \quad \mathbf{if}\ y \\ \qquad \mathbf{then}\ \mathbf{fun}\ k_f\ (w) = \mathcal{L}[\![exp_1]\!]k_{if}\ \mathbf{in}\ f\,(k_f) \\ \qquad \mathbf{else}\ \mathcal{L}[\![exp_2]\!]k_{if}\end{array}$$

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

Up to now, we have been passing the return continuation of an eligible function as an additional argument (as is standard in CPS conversion), but since our analysis has already told us that any eligible function f has exactly

$$\mathcal{L}[\![x]\!]\Diamond = x \tag{11}$$

$$\mathcal{L}[\![x]\!]k = k\,(x) \tag{12}$$

$$\mathcal{L}[\![\mathbf{fun}\ f\,(\vec{x}) = e_1\ \mathbf{in}\ e_2]\!]k = \begin{cases} \mathbf{fun}\ f\,(k',\,\vec{x}) = \mathcal{L}[\![e_1]\!]k'\ \mathbf{in}\ \mathcal{L}[\![e_2]\!]k \\ \quad \text{when } f \in \mathcal{E}\ (k' \text{ is fresh}) \\ \mathbf{fun}\ f\,(\vec{x}) = \mathcal{L}[\![e_1]\!]\Diamond\ \mathbf{in}\ \mathcal{L}[\![e_2]\!]k \\ \quad \text{otherwise} \end{cases} \tag{13}$$

$$\mathcal{L}[\![\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2]\!]k = \begin{cases} \mathbf{fun}\ k'\,(x) = \mathcal{L}[\![e_2]\!]k\ \mathbf{in}\ \mathcal{L}[\![e_1]\!]k' \\ \quad \text{when } \mathcal{S}(e_1) = \mathbf{true}\ (k' \text{ is fresh}) \\ \mathbf{let}\ x = \mathcal{L}[\![e_1]\!]\Diamond\ \mathbf{in}\ \mathcal{L}[\![e_2]\!]k \\ \quad \text{otherwise} \end{cases} \tag{14}$$

$$\mathcal{L}[\![\mathbf{if}\ x\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2]\!]k = \mathbf{if}\ x\ \mathbf{then}\ \mathcal{L}[\![e_1]\!]k\ \mathbf{else}\ \mathcal{L}[\![e_2]\!]k \tag{15}$$

$$\mathcal{L}[\![f\,(\vec{x})]\!]\Diamond = f\,(\vec{x}) \tag{16}$$

$$\mathcal{L}[\![f\,(\vec{x})]\!]k = \begin{cases} f\,(k,\,\vec{x}) \\ \quad \text{when } f \in \mathcal{E} \\ \mathbf{let}\ y = f\,(\vec{x})\ \mathbf{in}\ k\,(y) \\ \quad \text{otherwise } (y \text{ is fresh}) \end{cases} \tag{17}$$

$$\mathcal{S}(x) = \mathbf{false} \tag{18}$$

$$\mathcal{S}(\mathbf{fun}\ f\,(\vec{x}) = e_1\ \mathbf{in}\ e_2) = \mathcal{S}(e_2) \tag{19}$$

$$\mathcal{S}(\mathbf{let}\ x = e_1\ \mathbf{in}\ e_2) = \mathcal{S}(e_1) \text{ or } \mathcal{S}(e_2) \tag{20}$$

$$\mathcal{S}(\mathbf{if}\ x\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2) = \mathcal{S}(e_1) \text{ or } \mathcal{S}(e_2) \tag{21}$$

$$\mathcal{S}(f\,(\vec{x})) = \mathbf{true} \text{ if } f \in \mathcal{E} \text{ and } \mathbf{false} \text{ otherwise} \tag{22}$$

Figure 6. Local CPS conversion

one return continuation k_f, we can specialize the return sites of f to call k_f directly. To do so requires lifting the definition of k_f up to the binding site of f.[7] 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 we could allocate a closure for this purpose, it is more efficient to pass the free variables as function parameters as is done by *light-weight closure conversion* [30]. Having these variables as parameters in the final DS representation means that they will get mapped to MLRISC pseudo registers, which allows the register allocator to manage them. To illustrate closure conversion, consider the following fragment:

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

```
fun applyf (f, n) = lp_i (0, f, n)
and 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)
```

Figure 7. The `applyf` cluster after LCPS conversion

```
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:

```
fun f (y) = ... k_f(z, y)
and k_f (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, LCPS conversion 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 LCPS conversion (*i.e.*, all of its call sites have the same return continuation). The cluster resulting from applying the transformation to `lp_j` is given in Figure 7. We have used light-weight closure conversion on the internal fragments 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. Notice also that the cluster is still in a DS representation; for example, the call to `f` has an implicit return continuation.

5. Implementation

We have described LCPS as a transformation that is applied prior to the closure conversion, but in the MOBY compiler it is implemented as part of the cluster-conversion phase described in Section 2.2. In this section, we describe how our analysis and LCPS conversion are integrated into cluster conversion, and we also describe one other application of LCPS in our compiler.

5.1. ANALYSIS

The analysis is organized into two sub-phases. The first sub-phase constructs a *known-call graph* (*i.e.*, a partial call graph with edges for calls to known functions) and uses the graph to detect LCPS candidates and map functions to clusters. The second sub-phase determines the free variables of each cluster.

Computing the cluster assignment involves the following steps:

1. We construct the *known-call* graph from the BOL term. This graph has nodes corresponding to the functions defined in the BOL term, plus special *unknown caller* and *unknown callee* nodes. The graph has forward edges from callers to callees, which are labeled as corresponding to a tail or non-tail call, and back-edges from callees to callers, which are also labeled as to whether the corresponding call was a tail call. Either the callee or caller may be unknown.

2. The known-call graph is used to identify LCPS candidates by computing a conservative approximation of the return continuations of known functions. The analysis works by propagating abstract return continuations across the forward edges in the known-call graph. We handle mutually recursive functions by iterating to a fixed-point; since the analysis is monotonic and the abstract domain is short, this process converges quickly. This analysis produces a set of known functions whose non-tail call sites are candidates for LCPS conversion.

3. Frame conversion computes the cluster assignment using the known-call graph. We first determine those functions that must map to entry fragments; *i.e.*, those functions that either escape or have non-tail calls. The remaining functions are known functions that only have tail calls to them. If such a function is dominated by a single cluster, then it can be mapped to an internal fragment in that cluster. Otherwise, it must be mapped to an entry fragment of a new cluster. We use a depth-first traversal across the back edges in the known-call graph to compute this mapping.

4. If the LCPS analysis (Step 2 above) computes a non-empty set of candidate sites, then we determine *split-points* (*i.e.*, program points where a

new λ-abstraction must be introduced to explicitly represent a continuation). Split points correspond to the case in Rule 14 of Figure 6 where $\mathcal{S}(e_1) = \mathbf{true}$. We also determine the places where tail-calls to these continuations must be introduced.

5. The last step is to merge the cluster of the caller and callee for each LCPS candidate site. We also introduce fragments that correspond to the explicit continuation functions introduced by LCPS conversion.

In this process, the steps 2, 4, and 5 were added to support LCPS conversion. Once this sub-phase is complete, the mapping to clusters and fragments has been determined. It is represented using a combination of annotations on the BOL terms and auxiliary data structures.

The second part of the analysis is the *free variable analysis*. For each cluster, we compute its free variables, and for each fragment, we compute its *light-weight closure variables*. These variables are free variables of the fragment that defined elsewhere in the cluster, and thus can be passed in as extra arguments.

5.2. TRANSFORMATION

With the information computed by the analysis sub-phases, the transformation part of cluster conversion rewrites the BOL input term to BOL clusters. The transformation sub-phase also makes the representation of functions as code-pointer/environment-pointer pairs explicit by introducing code to allocate closures and replacing free variable references with fetches from the cluster's closure. Following cluster conversion, we have a pass that inlines continuation fragments that are only called once (for example, the continuation `k` in Figure 7 can be inlined in the body of `lp_j`).

5.3. DECONTIFICATION

Another application of LCPS conversion in the MOBY compiler is *decontification* — turning local continuation operations into tail function calls. BOL has a weak form of explicit continuations that is used to implement exceptions and concurrency primitives [7]. BOL continuations are not first-class, but instead have a lifetime that is restricted to the lifetime of their scope. This restriction makes it possible to stack allocate continuations. The binding

$$\textbf{let cont } \texttt{k}(x_1, \ldots, x_n) \texttt{ = } e_1 \textbf{ in } e_2 \textbf{ end}$$

reifies the current continuation by binding k to the current continuation prefixed by $\lambda\,(x_1, \ldots, x_n)\,.\,e_1$. The scope of k is e_2. The expression

$$\textbf{throw } \texttt{k } (x_1, \ldots, x_n)$$

transfers control to the continuation bound to k, with the values of the x_i bound to the parameters of k. For example, the expression

```
(let cont k(x) = x+1 in (throw k (7)) - 5 end) + 4
```

evaluates `12`.

As an example of how BOL continuations are used, consider the following MOBY function that wraps an exception handler around the call to `f`:

```
fun g (x : Int, y : Int) -> Int {
  (try f(x / y) except { ex => 17 }) + x
}
```

The compiler translates this code into the following BOL representation:

```
fun g (x, y, exh) = let
  val t1 = let
      cont handler (ex) = 17
      val t2 = if (y = 0)
        then throw handler (DivZeroExn)
        else x / y
      in f (t2, handler) end
  in t1 + x end
```

where `DivZeroExn` is a global BOL variable that is bound to the "division by zero" exception. In this translation, the continuation bound to `handler` is used both in a local **`throw`** and is passed as the exception handler to `f`.[8]

This example illustrates an opportunity for optimization. We would like the local **`throw`** to be compiled as a direct jump, but the continuation at the **`throw`** to `handler` is different from the continuation at `handler`'s binding. Our solution is to convert the right-hand-side of a **`cont`** binding into a tail-call of a new fragment and to use LCPS conversion to convert the enclosing continuation of the **`cont`** binding into another fragment. For the above example, this transformation results in the following BOL cluster (as before, we have omitted the explicit closure representations to make the example more readable):

```
fun g (x, y, exh) = let
  cont handler (ex) = k1 (17, x)
  in
    if (y = 0)
      then throw handler (DivZeroExn)
      else k1 (f (x / y, handler), x)
  end
and k1 (t1, x) = t1 + x
```

[8] When translating to BOL, we add an extra parameter to each function, which is its exception handler continuation. We omitted the handler parameters for the sake of simplicity in the previous examples.

Cluster conversion has introduced the fragment `k1` for the enclosing continuation of `handler` (`k1` is a join-point in the control-flow). The right-hand-side of the continuation binding is replaced by a tail-call (goto) to `k1` and the result of calling `f` is also passed to `k1`.

This code has the property that the **`throw`** to `handler` has the same continuation as at `handler`'s binding (*i.e.*, the **`throw`** is in a tail position), which means that we can correctly inline `handler` at the **`throw`** site.

```
fun g (x, y, exh) = let
  cont handler (ex) = k1 (17, x)
  in
    if (y = 0)
      then k1 (17, x)
      else k1 (f (x / y, handler))
  end
and k1 (t1, x) = t1 + x
```

We perform this transformation on all **`cont`** bindings and rely on a subsequent phase to inline **`throws`** to continuations that are bound in the same cluster.

6. Experience

To judge the usefulness of LCPS conversion, we measured the performance of three small programs compiled both with and without LCPS conversion. We disabled LCPS conversion by having the analysis return an empty set of LCPS candidates, which was the only difference between the two versions of the compiler. The programs we measured are:

`loop2` — a two-deep loop nest, where each loop is iterated 10000 times for a total of 10^8 iterations. This program is essentially the example given in Figure 1.

`loop3` — a three-deep loop nest,where each loop is iterated 1000 times for a total of 10^9 iterations.

`mm` — 100 iterations of the multiplication of two 100×100 integer matrices. We use a row-major representation of the matrices with explicit index computations and unchecked (or unsafe) array accesses.

The first two of these are synthetic benchmarks designed to highlight the impact of loop overhead. In these programs, the body of the inner loop is a call to an unknown null-function with the loop variables passed as arguments. Because the workload of these loops is minimal, the loop overhead dominates their execution time and any performance differences can be attributed to

Table I. Measured execution times (in seconds)

	Moby		OCaml		C
Benchmark	**(no LCPS)**	**(LCPS)**	**(no loops)**	**(loops)**	
`loop2`	2.15	1.83	2.08	1.81	1.54
`loop3`	23.64	20.63	22.21	20.87	16.79
`mm`	1.95	1.39	1.88	1.71	1.13

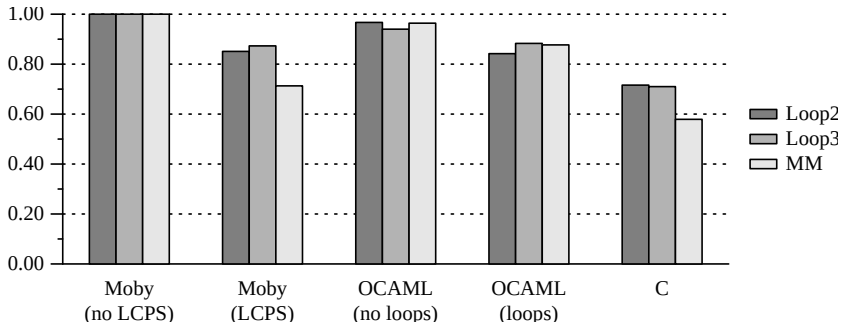

Figure 8. Normalized execution times

LCPS conversion. The third program is small, but represents a real-world application of nested loops.

We measured the wall-clock time (using the `time` command) to execute the program on a lightly-loaded 733MHz Pentium III system running version 2.2.14 of the Linux kernel. Each program was run five times and we report the median time. The raw performance data are given in Table I. We also normalized the execution time to the Moby version without LCPS conversion. The normalized times are shown in Figure 8; from this graph one can see that LCPS conversion reduces loop overhead in these cases from 10% to 25%. An examination of the generated machine code suggests several reasons for this improvement: the non-LCPS code allocates closure objects for the inner functions, which adds time to the loop and increases GC overhead; the non-LCPS code uses procedure call/return to implement the control-flow between the outer and inner loops; lastly, because LCPS conversion results in larger clusters, the register allocator is able to use machine registers more effectively for it. We expect to be able to increase the benefits of LCPS conversion by applying standard loop optimizations to the BOL cluster representation.

We also measured the execution time of optimized OCAML and C versions of these benchmarks.[9] The OCAML language 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 [18], so we measured OCAML versions of our benchmarks, written both with **`for`**-loops and with tail recursion. There are a couple of observations that we can make about the OCAML numbers. The tail-recursive MOBY code, when compiled with LCPS enabled, is competitive with the OCAML **`for`**-loop code (slightly slower on `loop2` and faster on `loop3` and `mm`), which demonstrates that one does not need special loop forms in a compiler's IR to support efficient loops. While special loop forms are not required in the IR, including them in the surface language may be desirable since they can provide a more concise notation for some computational structures. Unfortunately, OCAML exhibits a significant performance discrepancy between the two styles of writing nested loops, which means that programmers must consider the performance impact when choosing a programming style. By using LCPS, we can simplify the compiler (by eliminating the need for special looping forms in the IR), while providing consistent performance for both styles of programming nested loops. Since OCAML also uses a direct-style IR, it should be possible to improve its handling of nested recursion by adding LCPS conversion.

The C numbers are included as a sanity check; they demonstrate that while the MOBY/LCPS version is slower than C, it is a matter of less than 20% and not a factor of two. We believe that a large part of this difference is owed to the fact that the MLRISC framework does not do sophisticated code layout and jump chaining yet.

7. Related work

Most of the literature about compiler optimizations for strict functional languages uses CPS as a representation. Tarditi's thesis [33] 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 LCPS conversion (the OCAML [19], TIL [34, 33], and RML [23] compilers do not), but our techniques should be applicable to them. The MLJ compiler, which compiles SML to JAVA bytecodes performs a transformation that converts non-tail calls to gotos in a way that is similar to LCPS (see Section 6.6 of [3]), but the details of how and when this transformation is applied are not given.

[9] We used `ocamlopt` version 3.00 with the `-unsafe` option and `gcc` version 2.91.66 with the `-O2` option.

Kelsey describes a technique for merging functions in a first-order CPS-based representation [13, 14]. In his representation, λ abstractions are annotated with either *proc*, *cont*, or *jump* — λ_{proc} functions correspond to BOL clusters, λ_{jump} functions correspond to internal fragments, and λ_{cont} functions are return continuations. Kelsey outlines a scheme for merging clusters by converting a λ_{proc} to a λ_{jump} when the λ_{proc} is always called with the same return continuation. The MLton compiler for Standard ML uses a first-order representation, called FOL, that is similar to Kelsey's and it has a transformation phase, called *contification*, that performs a transformation similar to Kelsey's technique. Kelsey's paper did not describe an analysis for his transformation, but a recent paper by Fluet and Weeks describes the two analyses used in MLton [9]. The first of these, called $\mathcal{A}_{\text{Call}}$, detects the situation where Kelsey's transformation can be applied. It is similar to our frame conversion analysis (Step 3 on page 14). The second, $\mathcal{A}_{\text{Cont}}$, is an adaptation of our analysis to FOL. The major difference between our work and these schemes is that they are starting from a CPS representation where they already have all the explicit continuations they need and so their focus is on the cluster assignment problem, whereas we are focused on introducing a minimal number of explicit continuation functions in a DS representation to improve cluster merging. The fact that our analysis can be applied to a CPS IR to drive cluster merging is a result of the fact that it detects call sites that have a statically known return continuation.

LCPS is a special instance of the more general class of transformations, called *Selective CPS Transformation*, where a CPS conversion is applied partially rather than completely. One early example is work by Danvy and Hatcliff for CPS transforming call-by-name programs that have strictness annotations [5]. Their technique selectively uses the CBV CPS transformation for strict terms and the CBN CPS transformation for non-strict terms.

Kim, Yi, and Danvy have used selective CPS transformation as a technique for replacing SML's exception raising and handling mechanisms with continuation operations [16]. They introduce a pair of explicit return and handler continuations for each expression that might raise an uncaught exception and they add explicit continuation parameters to each function that might raise an uncaught exception. Their motivation is to allow optimizations similar to our decontification optimization described in Section 5.3. One major difference is that BOL already has an explicit representation of exception handler continuations, whereas Kim *et al.*'s work is primarily concerned with adding such continuations. Another difference is that while selective CPS conversion is a partial CPS conversion (*i.e.*, terms that cannot raise uncaught exceptions do not need explicit continuations), it is a global transformation that can span multiple functions. Our LCPS conversion, on the other hand, is local to one or two functions.

More recently, Kim and Yi have described a type-based approach for controlling partial CPS conversion [15]. Their motivating example is calling CPS-converted functions from non-CPS converted code, but one could probably use their annotated representation as a target of our analysis. A recent paper by Nielsen formally defines and proves correct a selective CPS conversion for a language with first-class continuations [21]. As with Kim *et al.*'s work, this technique requires global transformations.

8. Conclusion

We have presented a transformation, called Local CPS conversion, that can be used in a direct-style compiler to improve the performance of nested loops and have described its implementation in the MOBY compiler. Our measurements show a 10% to 25% reduction in loop overhead for a simple loop nests. While we have only presented fairly simple examples, LCPS conversion can handle complicated looping structures, such as multiple inner loops and loops expressed as mutually recursive functions as one would get when encoding state machines. LCPS also enables specializing returns when the return context is known (*i.e.*, there is only one return continuation) [28]. We have also sketched another application of LCPS conversion: *decontification*, where we replace **`throws`** with direct jumps (represented as tail calls). LCPS conversion 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

Olivier Danvy, Kathleen Fisher, Lal George, Jon Riecke, and the anonymous referees provided many useful comments on various drafts of this paper. Kathleen Fisher helped debug the implementation and Allen Leung provided timely MLRISC improvements and bugfixes. Stephen Weeks provided a detailed description of the MLton contification analysis and transformation.

References

1. Appel, A. W.: 1992, *Compiling with Continuations*. New York, N.Y.: Cambridge University Press.
2. Appel, A. W. and T. Jim: 1997, 'Shrinking lambda expressions in linear time'. *Journal of Functional Programming* **7**(4), 515–540.
3. Benton, N., A. Kennedy, and G. Russell: 1998, 'Compiling Standard ML to Java bytecodes'. In: *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*. pp. 129–140.

4. Damian, D. and O. Danvy: 2000, 'Syntactic accidents in program analysis: On the impact of the CPS transformation'. In: *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. pp. 209–220.
5. Danvy, O. and J. Hatcliff: 1992, 'CPS transformation after strictness analysis'. *ACM Letters on Programming Languages and Systems* **1**(3), 195–212.
6. Fisher, K. and J. Reppy: 1999, 'The design of a class mechanism for Moby'. In: *Proceedings of the SIGPLAN'99 Conference on Programming Language Design and Implementation*. pp. 37–49.
7. Fisher, K. and J. Reppy: 2001, 'Compiler support for lightweight concurrency'. Submitted for publication (available from `www.cs.bell-labs.com/~jhr/moby`).
8. Flanagan, C., A. Sabry, B. F. Duba, and M. Felleisen: 1993, 'The essence of compiling with continuations'. In: *Proceedings of the SIGPLAN'93 Conference on Programming Language Design and Implementation*. pp. 237–247.
9. Fluet, M. and S. Weeks: 2001, 'Contification using dominators'. In: *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*. pp. 2–13.
10. George, L. and A. Appel: 1996, 'Iterated register coalescing'. *ACM Transactions on Programming Languages and Systems* **18**(3), 300–324.
11. George, L., F. Guillaume, and J. Reppy: 1994, 'A portable and optimizing back end for the SML/NJ compiler'. In: *Fifth International Conference on Compiler Construction*. pp. 83–97.
12. Johnsson, T.: 1985, 'Lambda-lifting: Transforming programs to recursive equations'. In: *Functional Programming Languages and Computer Architecture*. pp. 190–203.
13. Kelsey, R. and P. Hudak: 1989, 'Realistic compilation by program transformation'. In: *Conference Record of the 16th Annual ACM Symposium on Principles of Programming Languages*. pp. 281–292.
14. Kelsey, R. A.: 1995, 'A correspondence between continuation passing style and static single assignment form'. In: *Proceedings of the ACM SIGPLAN Workshop on Intermediate Representations*. pp. 13–22.
15. Kim, J. and K. Yi: 2001, 'Interconnecting between CPS terms and non-CPS terms'. in [25], pp. 7–16.
16. Kim, J., K. Yi, and O. Danvy: 1998, 'Assessing the overhead of ML exceptions by selective CPS transformation'. In: *Proceedings of the 1998 ACM SIGPLAN Workshop on ML*. pp. 103–114.
17. Kranz, D., R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams: 1986, 'Orbit: An optimizing compiler for Scheme'. In: *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction*. pp. 219–233.
18. Leroy, X.: 1997, 'The effectiveness of type-based unboxing'. In: *ACM SIGPLAN Workshop on Types in Compilation (TIC97)*. Available as Technical report BCCS-97-03, Boston College, Computer Science Department.
19. Leroy, X.: 2000, 'The Objective Caml System (release 3.00)'. Available from `http://caml.inria.fr`.
20. Moggi, E.: 1991, 'Notions of computation and monads'. *Information and Computation* **93**(1), 55–92.
21. Nielsen, L. R.: 2001, 'A selective CPS transformation'. In: S. Brooks and M. Mislove (eds.): *Electronic Notes in Theoretical Computer Science*, Vol. 45. Also available as BRICS Report RS-01-30.
22. Nielson, F., H. R. Nielson, and C. Hankin: 1999, *Principles of Program Analysis*. New York, NY: Springer-Verlag.
23. Oliva, D. P. and A. P. Tolmach: 1998, 'From ML to Ada: Strongly-typed language interoperability via source translation'. *Journal of Functional Programming* **8**(4), 367–412.

24. Reppy, J.: 2001, 'Local CPS conversion in a direct-style compiler'. in [25], pp. 13–22.
25. Sabry, A. (ed.): 2001, 'Proceedings of the Third ACM SIGPLAN Workshop on Continuations (CW'01)'. Technical Report 545, Computer Science Department, Indiana University.
26. Sabry, A. and M. Felleisen: 1994, 'Is Continuation-passing style useful for Data Flow analysis?'. In: *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*. pp. 1–12.
27. Shao, Z. and A. W. Appel: 1994, 'Space-efficient closure representations'. In: *Conference record of the 1994 ACM Conference on Lisp and Functional Programming*. pp. 150–161.
28. Shao, Z., J. H. Reppy, and A. W. Appel: 1994, 'Unrolling Lists'. In: *Conference record of the 1994 ACM Conference on Lisp and Functional Programming*. pp. 185–191.
29. Shivers, O.: 1991, 'Control-flow Analysis of Higher-order Languages or Taming Lambda'. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. Available as Technical Report CMU-CS-91-145.
30. Steckler, P. A. and M. Wand: 1997, 'Lightweight Closure Conversion'. *ACM Transactions on Programming Languages and Systems* **19**(1), 48–86.
31. Steele Jr, G. L.: 1976, 'LAMBDA: The ultimate declarative'. Technical Report AI Memo 379, Massachusetts Institute Technology, Artificial Intelligence Laboratory.
32. Steele Jr., G. L.: 1978, 'Rabbit: A compiler for Scheme'. Master's thesis, MIT.
33. Tarditi, D.: 1996, 'Design and implementation of code optimizations for a type-directed compiler for Standard ML'. Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. Available as Technical Report CMU-CS-97-108.
34. Tarditi, D., G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee: 1996, 'TIL: A type-directed compiler optimizing compiler for ML'. In: *Proceedings of the SIGPLAN'96 Conference on Programming Language Design and Implementation*. pp. 181–192.