-

Continuation-Passing, Closure-Passing Style

*Andrew W. Appel**
Trevor Jim†

CS-TR-183-88

July 1988
Revised September 1988

ABSTRACT

We implemented a continuation-passing style (CPS) code generator for ML. Our CPS language is represented as an ML datatype in which all functions are named and most kinds of ill-formed expressions are impossible. We separate the code generation into phases that rewrite this representation into ever-simpler forms. Closures are represented explicitly as records, so that closure strategies can be communicated from one phase to another. No stack is used. Our benchmark data shows that the new method is an improvement over our previous, abstract-machine based code generator.

To appear in POPL ’89.

* Supported in part by NSF Grant DCR-8603543 and by a Digital Equipment Corp. Faculty Incentive Grant.

† AT&T Bell Laboratories, Murray Hill, NJ. Current address: Laboratory for Computer Science, MIT, Cambridge, Mass.

1. Overview

Standard ML of New Jersey[1] is a compiler for ML written in ML. Its first code generator, based on an abstract stack machine, produced code with acceptable but not stunning performance. Examination of the code revealed that the greatest source of inefficiency seemed to be that each value went on and off the stack too many times. Rather than hack a register allocator into the abstract stack machine, we decided to try a continuation-passing style (CPS)[2] code generator. Kranz's ORBIT compiler[3] [4] shows how CPS provides a natural context for register allocation and representation decisions.

The beauty of continuation passing style is that control flow and data flow can be represented in a clean intermediate language with a known semantics, rather than being hidden inside a ''black box'' code generator. The ORBIT compiler translates CPS into efficient machine code, making representation decisions for each function and each variable. ORBIT does an impressive set of analyses in its back end, but they're all tangled together into a single phase. We have a series of phases, each of which re-writes and simplifies the representation of the program, culminating in a final instruction emission phase that's never presented with complications.

The phases are:

1. Lexical analysis, parsing, typechecking, producing an annotated abstract syntax tree.
2. Translation into lambda-calculus (producing a simple representation described in[1]).
3. Optimization of the lambda-calculus (present here for historical reasons; this phase duplicates some of the effort done by our CPS optimizer).
4. Conversion into continuation-passing style, producing a CPS representation described in the next section.
5. Optimization of the CPS expression.
6. Closure conversion, producing a CPS expression in which each function is closed (i.e. has no free variables).
7. Elimination of nested scopes, producing a CPS expression with one global set of mutually-recursive, non-nested function definitions.
8. ''Register spilling,'' producing a CPS expression in which no sub-expression has more than *n* free variables, where *n* is related to the number of registers on the target machine.
9. Generation of target-machine instructions.
10. Backpatching and jump-size optimization.

Where the ORBIT compiler has one black box covering phases 6 through 9, we have four smaller black boxes. The interfaces between the phases are semantically well-defined, making it easier to isolate individual parts of the analysis to one phase.

This paper describes phases 4 through 9, and then presents an analysis based on profiling and benchmarks. Because of space limitations, we must assume that the reader is familiar with continuation-passing style.

2. Continuation-passing style

Our back-end representation language is a continuation-passing style (CPS) representation similar in spirit to Steele's, but with a few important differences: we use the ML datatype feature to prohibit ill-formed expressions; we want every function to have a name; and we have *n*-tuple operators which make modelling closures convenient.

An important property of well-formed CPS expressions in Steele's representation is that a function-application can never be the direct child of another application. We can express this restriction directly in the ML datatype *cexp* (for *continuation-expression*):

```
datatype cexp
= RECORD of
     var list * var * cexp
| SELECT of
     int * var * var * cexp
| APP of
     var * var list
| FIX of
   (var * var list * cexp) list * cexp
| SWITCH of
   var * cexp list
| PRIMOP of
  int * var list * var list * cexp list
```

The italicized *var*'s are binding occurrences, and the others are uses of the variables.

All of Steele's ''atoms''[2] are represented in our cexp's by variables. Constants are represented by globally-free variables entered in an auxiliary table. This means that a function application (`APP`) can be represented by the name of the function (*var*) and a series of arguments (*var list*). The constraint that an `APP` can't be a child of an `APP` is enforced by the fact that the arguments of the `APP` constructor are variables, not continuation-expressions.

One of the useful properties of CPS is that every intermediate value of a computation is given a name. In Steele's representation, however, functions can still be anonymous, making it difficult for a code generator to keep track of them. Therefore, we eliminate LAMBDA from our CPS datatype, in favor of `FIX`, a general-purpose mutually recursive function definition in which names are explicitly bound to functions:

```
FIX([(f,[x,y,z],
        (...x...g...y...z...f...)),
     (g,[i,j],
        (...j...i...g...f...g...)) ],
    ...g...f... )
```

(In ML notation, [x,y,z] is a list of three elements and (a,b,c) is a tuple of three elements.) This example defines two mutually recursive functions *f* and *g* of two and three arguments, respectively; the ellipses are other expressions, and the entire example is an expression. The binding is roughly equivalent to the ML expression:

```
let fun f(x,y,z) =
         ...x...g...y...z...f...
    and g(i,j) =
         ...j...i...g...f...g...
 in ...g...f...
end
```

Function definitions can be nested in other expressions; expressions (and functions) can, of course, have free variables bound at outer levels.

`RECORD` and `SELECT` are used to manipulate *n*-tuples. `RECORD([a,b,c,d],r,cont)` means ''let $r=(a,b,c,d)$ in *cont*'', and `SELECT(i,r,v,cont)` means ''let *v* be the i^{th} field of *r* in *cont*''.

We have a constructor for indexed jumps (`SWITCH`), and a constructor (`PRIMOP`) for miscellaneous in-line primitive operations like integer and floating arithmetic, array subscript, etc. The expression

```
PRIMOP(i,[a,b,c],[d,e],[F,G,H])
```

means to apply operator *i* to the arguments (a,b,c) yielding the results *d* and *e*, and then branch to one of the continuations *F*, *G*, or *H*. Each primitive operator has its own ''signature;'' for example

```
PRIMOP(plus,[a,b],[d],[F])
```

takes two arguments, returns one result, and continues in only one way, whereas

```
PRIMOP(lessthan,[a,b],[],[F,G])
```

takes two arguments, produces no result, and branches to *F* or *G*.

One goal in choosing our representation is to give each object a name (i.e. a variable). Why did we not represent the control-flow branches of a `SWITCH` or `PRIMOP` by variables standing for continuation functions? The problem is with the free variables of the different control-flow branches. In the CPS language as shown in this section, any sub-expression and any (nested) function may have free variables (that are bound at an outer level of nesting). However, in one of our code generation phases we will rewrite the CPS graph to eliminate free variables from all functions. If the control-flow branches are to be represented as functions, then they could not have free variables without creating extra closures; in fact, it would be necessary to create closures for all branches even though only one branch would be taken. Therefore we compromise and leave the branches as unnamed continuation expressions instead of named continuation variables.

3. Conversion into CPS

The front end of our compiler produces a lambda-calculus intermediate representation (described in [1]). This must be translated into continuation-passing style; the conversion algorithm is similar to Steele's and won't be described in detail here.

The conversion process doesn't do many optimizations; it's simpler to do that in a separate phase. The converter has its hands full just with the semantics of the two languages (lambda-calculus and continuation-passing style) that it is translating between. It does make these representation decisions:

- Makes control flow explicit by the use of continuations.
- ''Lowers'' typed constructs like ML's disjoint union constructors into untyped constructs like `RECORD`s with integer tags.
- Optimizes the representation of case statements (arising from ML pattern-matching) into jump-tables or binary trees of comparisons.[5].
- The pattern $(\lambda x.\ M)(N)$, which has the effect of let $x = N\ \ in\ \ M$, is treated specially. This is an optimization that could be left for the next phase, but it is convenient and cost-effective to recognize it here.

4. Reduction of the CPS

The next phase is a CPS ''reducer'' that performs a variety of optimizations. They are listed here, each with an indication of how often it is applicable for each 1000 operands* of CPS graph:

205: Replace `SELECT(i,r,...)` with the i^{th} field of r, when r is a statically determinable record.

181: Perform beta-reduction (inline expansion) on any function that is called only once, or whose body is not too large.

72: Merge sets of mutually recursive function definitions (`FIX`es) in the hope that they will later share the same closure. Merging can be done if one FIX is the immediate child of another, and each has the same set of free variables.

66: Perform eta-reduction (where f(x,y)=g(x,y), replace all uses of f with g).

47: Perform constant-folding on `SWITCH`es and `PRIMOP`s.

26: Hoist (un-nest, or enlarge the scope of visibility of) function definitions to enable the merging of `FIX`es.

4: Remove unused arguments of functions.

2: Flatten the arguments of (nominally single-argument) ML functions that are always called with a tuple of actual parameters.

0.1: Remove the definitions of variables that aren't used.

Our optimizer makes several passes (typically half a dozen) before no (or few) redexes remain. The test that produced the frequencies above counted all passes on a 16,000-line ML program that had a graph of size 118514. If the optimizer were to stop at module boundaries, the numbers would be somewhat different. Our compiler, for historical reasons, also has an optimizer in the (non-CPS) lambda-calculus level, which has some overlap with the CPS reducer and also will affect the counts given here.

5. Closure conversion

When one function is nested inside another, the inner function may refer to variables bound in the outer function. A compiler for a language where function nesting is permitted must have a mechanism for access to these variables. The problem is more complicated in languages (like ML) with higher-order functions, where the inner function can be called after the outer function has returned.

The usual implementation technique uses a ''closure'' data structure: a record containing the free variables of the inner function as well as a pointer to its machine code. A pointer to this record is made available to the machine code while it executes so that the free variables are accessible. By putting the code-pointer at a fixed offset (e.g. 0) of the record, users of the function need not know the format of the record or even its size in order to call the function.

In fact, several functions can be represented by a single closure record containing the union of their free variables and code pointers. A closure record is necessary only for a function that ''escapes'' — some of its call sites are unknown because it is passed as an argument, stored into a data-structure, or returned as a result of a function-call. A call of an escaping function is implemented by extracting the code pointer from the closure record and jumping to the function with the closure record as one of its arguments.

* This is a larger and more useful quantity than the number of nodes in the graph.

The closure of a ‘‘known’’ function (whose every call site is known at compile-time) need not be implemented as a record. Instead, the free variables can be added as extra arguments to the function. A call of a known function must arrange to pass along the appropriate variables to the function. This implementation of closures is intended to produce efficient code for loops.

Some functions escape and are also called from known sites. These can be split into two functions, where the escaping function is defined in terms of the known one, in the hope that the known calls will execute more efficiently.

Our closure converter rewrites a CPS expression, making function-closure representation explicit; we call this explicit representation ‘‘closure-passing style.’’ After a pass to gather free variable information,* the converter traverses the CPS expression. At every `FIX` that binds an escaping function, a `RECORD` is inserted to create an explicit closure record, and a new argument corresponding to the closure record is added to the argument list of each function.† Free variables accessed within the body of the function are rewritten as `SELECT`s from this argument.

Known-function bindings are rewritten by adding an argument for each free variable in the function’s body. Applications of the function are rewritten as the application to the arguments and the necessary free variables.

The following code fragment shows a sample ML function and the transformations applied to it. In rewriting a function *f*, our convention is to call the new, closed function *f′*, its closure record (if any) *f*, and the formal parameter corresponding to the closure record *f′′*. The first element of a closure record *f* will be *f′*, so that if *f* escapes to some context where *f′* is not known, the code-pointer *f′* can be `SELECT`ed from the closure record. All other references to escaping functions become references to closure records.

*Mutually-recursive functions complicate the free variable analysis, but this turns out to be a classical dataflow problem (live-variable analysis) that can be solved by classical techniques.[6]

† This technique has been used in the Categorical Abstract Machine[7]

```
fun f x =
  let fun g y = x+y
  in  g z
  end

 ...f 3...
```

in CPS:

```
FIX([(f,[x,c1],
      FIX([(g,[y,c2],
            PRIMOP(+,[x,y],[a],
           [APP(c2,a)]))]
          APP(g,[z,c1]))))],

    ...APP(f,[3,c0])...)
```

after closure conversion:

```
FIX([(f’,[f’’,x,c1],
      FIX([(g’,[y,c2,x],
            PRIMOP(+,[x,y],[a],
           [APP(c2,a)]))]
       SELECT(1,f’’,z,
             APP(g’,[z,c1,x]))))],
    RECORD([f’,z],f,

    ...SELECT(0,f,f’
         APP(f’,[f,3,c0])...)
```

The function *g* is known, so its free variable *x* has been added to its argument list; function *f* escapes, and requires a closure record. See the appendix for a larger example written in a more readable notation.

In more complicated examples, involving many variables from differing scopes, there can be a number of possible closure representations for a function.[8] One simple strategy is to use a flat closure containing all the free variables. At the other extreme, a number of closure records already exist when a new closure must be created, and a pointer or link to one can provide access to several of the necessary variables. Combinations of the two allow us to trade off time of closure creation, size of closures, and ease of access to variables from closures. The tradeoffs can be subtle: for example, linked closures can take up more space than flat closures because they hold on to closures which might otherwise be reclaimed by the garbage collector. Several strategies have been implemented in Standard ML of New Jersey.

6. Flattening and spilling phases

After closure conversion, functions no longer refer to non-constant free variables; therefore, nesting of function definitions is not necessary. A simple flattening pass gathers all the function definitions of a compilation unit into a single set of mutually-recursive function declarations; each declaration will correspond to a code fragment in the final machine code. Gathering the fragments lets us generate code for functions in any order, which helps make calls of known functions more efficient (see section 7).

Next, a register-spilling phase rewrites the CPS expression so that there are no more than *n* free variables at any subexpression, where *n* is related to the number of general purpose registers on the target machine. Wherever the number of free variables at a subexpression is larger than *n*, a `RECORD` containing some of the free variables is inserted. The appropriate `SELECT` is inserted at uses of those variables in the rest of the expression.

For example, consider a program that fetches six values from an array and adds them:

```
PRIMOP(subscr,[arr,1],a,
PRIMOP(subscr,[arr,2],b,
PRIMOP(subscr,[arr,3],c,
PRIMOP(subscr,[arr,4],d,
PRIMOP(subscr,[arr,5],e,
PRIMOP(+,[a,b],[g],[
PRIMOP(+,[g,c],[h],[
PRIMOP(+,[h,d],[i],[
PRIMOP(+,[i,e],[j],[ ...
```

There are five variables free in the first plus-expression. If compiling for a machine with only four registers, we can limit the number of simultaneous free variables by packing a,b,c into a record r and d,e into a record s:

```
PRIMOP(subscr,[arr,1],a,
PRIMOP(subscr,[arr,2],b,
PRIMOP(subscr,[arr,3],c,
RECORD([a,b,c],r,
PRIMOP(subscr,[arr,4],d,
PRIMOP(subscr,[arr,5],e,
RECORD([d,e],s,
SELECT(0,r,a',
SELECT(1,r,b',
PRIMOP(+,[a',b'],[g],[
SELECT(2,r,c',
PRIMOP(+,[g,c'],[h],[
SELECT(0,s,d',
PRIMOP(+,[h,d'],[i],[
SELECT(1,s,e',
PRIMOP(+,[i,e'],[j],[ ...
```

Now each sub-expression of this continuation-expression has only three variables free. The discerning reader will note that we could have rearranged the additions and the subscripts and avoided the use of records entirely, but some primops have side-effects that prevent rearrangements. Furthermore, spilling is rarely needed, so that improvements to the spilling algorithm won't affect overall performance much.

The spiller could have been combined with the closure converter; this would let us apply the closure representation analysis to spill records. However, spilling's rarity and the extra complexity required to combine the phases convinced us to implement spilling separately.

Our method of spilling has two important consequences. First, `PRIMOP`s and `APP`s must have no more than *n* arguments (where *n* is the number of registers on the machine). Thus we make sure that the optimizer never flattens the arguments of a known function if it would cross the limit, and that the closure converter implements known functions with more than *n* arguments and free variables by packaging the free variables into a closure record.

Second, it means that the CPS datatype described in section 2 does not allow the creation of records from more than *n* free variables. In practice, we use a `RECORD` constructor with (*var* * *path*) elements. The var can be a spill record, in which case the path specifies an element to select from it. Large records are made by computing some of the elements, spilling them, computing more, and eventually constructing the final record from variables in both spill records and registers. Although this might seem expensive, profiling shows that all spill records take up only one or two percent of the total heap allocation in our Vax implementation, where *n* is 8.

7. Generation of target-machine instructions

Since modern garbage collectors are so cheap[9] [10] we have dispensed with the stack. This simplifies the code generator, which doesn't need to do the analysis[2] [4] [11] necessary to decide which closure records can be allocated on the stack; it also simplifies the runtime system, making it easier to add multiple threads or state-saving operators to the programming environment.

Eliminating the stack is advantageous not only because it makes the compiler simpler. Operations like call-with-current-continuation (which, though not in ML, is compatible with the ML type system) are more efficient if there is no stack; a generational garbage collector can traverse just the newest call-frames, whereas it would have to traverse all the call frames on a stack; and in a multi-thread environment with stacks, a large stack space must be allocated for each thread even if it won't all be used.

The expression handed to the target-machine instruction generator has a very simple form indeed. Procedures never return (as a result of CPS conversion), procedures don't have non-constant free variables (as a result of closure analysis), scopes aren't nested, and there are never more live variables than registers to hold them.

Since all representation decisions have been made in previous phases, the decisions made by the instruction generator have mostly to do with register allocation. As an example, consider the fragment `SELECT(3,v,w,`*cexp*`)`, which requires that the third field of the record *v* be fetched into the (newly-defined) variable *w*, and execution to continue with the expression *cexp*. Both *v* and *w* can be allocated to machine registers, as a result of the (previous) spilling analysis. The variable *v* will have already been allocated at the time it was bound in the enclosing expression. The variable *w* must be allocated at this time to a register. Several heuristics are used.

Two-address instructions:

> Some instructions on some machines prefer to have their result argument in the same register as a source argument (this preference doesn't typically apply to fetches, so it won't probably wouldn't be used for this `SELECT`).

Targeting:

> Sometimes there's an opportunity to avoid a *move* instruction later on. If the variable *w* is used (in *cexp*) as the n^{th} argument to a function *f* whose calling sequence requires the n^{th} argument in register *r*, and if *r* is not bound to any other live variable, then *r* should be used for *w* to save the cost of a move instruction when *f* is called.

Anti-targeting:

> If there is a call in *cexp* to a function *f*, one of whose arguments (which is not *w*) is to be passed in register *r*, then *r* is to be given less preference than another register, to avoid the cost of moving *w* out of the way when *f* is called.

Default:

> Otherwise, any register not already allocated to a live variable may be used. The work done by the spiller ensures that there will always be a register available.

As in the ORBIT compiler, the instruction generator treats ''known'' functions (those whose call sites are all known statically) specially. The parameters of a known function can be allocated to registers in a way that optimizes at least one of the calls to the function. Specifically, the code for a known function is not generated until a call site is found and generated. Then, the formal parameters of the function can be allocated to the same registers where the call's actual parameters are already sitting; and the transfer of control can be by ''falling through'' without a jump. Thus, at least one call to each known function can be at no cost (except for actual parameters that are constants, and must be fetched into registers).

8. Benchmarks

We ran five different compilers on five different benchmark programs, all on a VAX 8650. The compilers were:

Pascal	Berkeley Pascal with -O option
ORBIT	Version 3.0 of the T system from Yale, with the ORBIT code generator.

Old — Our old code generator for ML, based on an abstract stack machine.

CPS — Our new code generator, described in this paper.

CPS’ — Our new code generator with aggressive cross-module optimization enabled.

Of course, comparisons between compilers for different programming languages may tell us more about the languages than about the compilers.

The programs were:

Hanoi — The towers of Hanoi benchmark from Kranz’s thesis[4].

Puzz — A compute-bound program from Forest Baskett[4].

LenL — A tail-recursive function (or, in Pascal, a while loop) to compute the length of a list.

LenR — A recursive function (not tail-recursive) to compute the length of a list.

Comp — A 16000-line compilation job in Standard ML. This is intended to measure the performance of real systems, not just artificial benchmarks.

	Hanoi	Puzz	LenL	LenR	Comp
Pascal	.42	2.02	1.43	7.52	
ORBIT	˜.4	˜2.1	.9	3.6	
Old	1.28	8.81	5.62	5.71	1613
CPS	.72	2.63	1.18	3.89	1432
CPS’	.21	2.87	1.09	4.53	1224

The table above gives execution times in seconds, not including garbage-collection overhead (which can be arbitrarily large or small depending on memory size[10]).

9. Results

By separating the code generation into easily-understood phases with clean interfaces, we make it easier to produce robust optimizing compilers. Our method is not difficult to implement, and works well in practice.

Our CPS code generator produces code that runs up to four times faster than our old, stack-based code generator on small benchmarks, and seems comparable to Pascal and ORBIT for the examples we tested. But on the large, ‘‘real world’’ benchmark, our CPS code generator did only 25% better than the old one. The reason seems to be that though the new code generator produces very efficient code for tight, tail-recursive loops, big programs tend to have more function calls requiring saving of state.

It might be argued that since we save state by making continuation closure records, it is our stackless strategy that slows performance. However, we estimate that even if every closure record were stack allocated we would save only 6% to 10%. Furthermore, the stackless strategy tends to use less memory (typically on the order of 20%, but sometimes a much greater savings) than the old, stack-based code, because objects tend to be retained on the stack after their last use. And the stackless strategy has other advantages, like a simpler runtime system and garbage collector.

Acknowledgements

David MacQueen was co-designer and co-implementor of the front end of the ML compiler. David Kranz made many useful suggestions about benchmarks.

Appendix: An example in detail

To illustrate the several phases of the code generation, we show the transformations made to a fragment of an ML program. The program has function, `count`, that takes a predicate (a function from α to boolean, for some type α) as an argument, and returns a function that counts how many elements of a list (of α) satisfy that predicate. Then a function `countZeros` is made by applying `count` to a predicate that returns `true` on 0 and false on other integers:

```
fun count(pred) =
 let fun f(x::rest) =
           if pred(x)
            then 1+f(rest)
            else f rest
        | f nil = 0
  in f
 end

val countZeros =
    count (fn 0 => true
            | _ => false)
```

This function will be translated by the compiler into lambda-calculus and then into CPS, but for illustrative purposes we will show all the transformations using the syntax of ML. The first transformation is in the front end of the compiler, where pattern-matches are converted into decision trees (i.e. if-expressions):

```
fun count(pred) =
  let fun f(a) =
          if null(a)
           then 0
           else if pred(a.0)
                 then 1+f(a.1)
                 else f a.1
   in f
  end

val countZeros =
   count (fn i => if i=0
                   then true
                   else false)
```

The next transformation is the conversion into continuation-passing style: each function gets an additional continuation argument and ‘‘returns’’ by calling the continuation:

```
fun count(pred,c1) =
 let fun f(a,c2) =
      if null(a)
       then c2(0)
       else
        let fun c3(b) =
            if b
              then
               let fun c4(i)=c2(1+i)
                in f(a.1,c4)
               end
              else
               let fun c5(i)=c2(i)
                in f(a.1,c5)
               end
        in pred(a.0, c3)
       end
  in c1(f)
 end

val countZeros =  . . .
```

The next phase is the optimization phase. Here there is one η-reduction (the removal of the function `c5`) and several β-reductions that can be done. In particular, the function `count` is not recursive (even though it contains a recursive function nested within it), and is ‘‘small’’ enough that the optimizer decides to put it ‘‘inline’’ inside `countZeros`. The function `pred` can then be expanded inside the copy of `count`.

```
fun count (pred,c1) = . . .

fun countZeros(m,c0) =
 let fun f(a,c2) =
      if null(a)
       then c2(0)
       else if a.0 = 0
            then
           let fun c4(i)=c2(1+i)
                in f(a.1,c4)
               end
            else f(a.1,c2)
  in f(m,c0)
 end
```

Now comes the closure conversion phase. The function `countZeros` ‘‘escapes’’ — it can be called by functions that don’t know its structure — so it must use the standard calling sequence. The standard calling sequence has three arguments: closure record, user-argument, and continuation. Here, the function has no free variables, but a closure record must still be built, since the function is first-class (i.e. escapes).

The function `f` doesn’t escape, and doesn’t happen to need a closure. The function `c4` escapes (it is an argument to a function), and needs a closure anyhow because it has a free variable `c2` (renamed c7 here). Continuations have a two-argument standard calling sequence (different from the three-argument standard for user-functions, since continuations don’t themselves have a continuation argument). The closure for `c4` will be called `c4`, while the function that implements it will be `c4’`, and the argument in which the closure record is supposed to be passed is called `c4’’`.

```
fun countZeros(e0,m,c0) =
 let fun f(a,c2) =
   if null(a)
   then c2.0(c2,0)
   else if a.0 = 0
        then let fun c4’(c4’’,i) =
                  let val c7=c4’’.1
                   in c7.0(c7,1+i)
                  end
                val c4 = [c4’,c7]
             in f(a.1,c4)
            end
        else f(a.1,c2)
  in f(m,c0)
 end
```

The next phase is trivial: since functions no longer have free variables, they can be un-nested.

```
fun countZeros(e0,m,c0) = f(m,c0)
and f(a,c2) =
     if null(a)
       then c2.0(c2,0)
       else if a.0 = 0
             then let val c4=[c4',c2]
                   in f(a.1,c4)
                  end
             else f(a.1,c2)
and c4'(c4'',i) =
        let val c7 = c4''.1
         in c7.0(c7,1+i)
        end
```

The spilling phase won't change this program, since there are never very many variables free at the same point. The next phase, register allocation and code generation (for the VAX, in this example), has more to do. Since `countZeros` escapes, it must be given a standard calling sequence using registers 0, 1, and 2 for the closure, argument, and continuation. Two-argument standard functions (e.g. the continuation `c4'`) have a different calling sequence using registers 2 and 1 for closure and argument; we arrange it this way because the continuation of a function will typically be the same as the closure of the next continuation called, and it will already be sitting in the right register. The function `f` may have any calling sequence, and this is chosen on the first call so that `countZeros` won't have to shuffle any registers.

As described in [1], integers are represented with a low-order 1 bit, pointers with a low-order zero. Allocation (e.g. for the closure after L1 below) is in-line, relying on a page fault to tell it when it needs to garbage-collect; register 12 points to the next available allocable space.

This program would be more efficient (and would not allocate closures) if it had been written tail-recursively, but even as it is, it will run very quickly:

```
countZeros:               e0 in r0, m in r1, c0 in r2
                          fall through:  f(m,c0)
f:                        a in r1, c2 in r2
 blbc  r1,L1              branch if a is not null
 clrl  r1                 arg0 in r2, arg1(=0) in r1
 movl  (r2),r0            r0 = c2.0
 jmp   (r0)               c2.0(c2,0)
L1:                       else clause
 movl  (r1),r0            r0 = a.0
 cmpl  $1,r0              test for "zero"
 jne   L2                 branch to else clause
 movl  r2,4(r12)          second field of new record
 moval c4',(r12)          make record [c4',c2]
 movl  $0x21,-4(r12)      descriptor of record
 movl  r12,r2             r2 = c4
 addl2 $12,r12            allocation bookkeeping
 movl  4(r1),r1           r1 = a.1
 jbr   f                  f(a.1,c4)
L2:                       second else clause
 movl  4(r1),r1           r1 = a.1
 jbr   f                  f(a.1,c2)
c4':                      r1 = i, r2 = c4''
 addl2 $2,r1              r1 = 1+i
 movl  4(r2),r2           r2 = c7
 movl  (r2),r0            r0 = c7.0
 jmp   (r0)               c7.0(c7,1+i)
```

References

1. Andrew W. Appel and David B. MacQueen, ‘‘A Standard ML compiler,’’ in *Functional Programming Languages and Computer Architecture (LNCS 274)*, pp. 301-324, Springer-Verlag, 1987.
2. Guy L. Steele, ‘‘Rabbit: a compiler for Scheme,’’ AI-TR-474, MIT, 1978.
3. D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams, ‘‘ORBIT: An optimizing compiler for Scheme,’’ *Proc. Sigplan ’86 Symp. on Compiler Construction*, vol. 21 (Sigplan Notices), no. 7, pp. 219-233, July 1986.
4. David Kranz, ‘‘ORBIT: An Optimizing Compiler for Scheme,’’ PhD Thesis, Yale University, 1987.
5. Andrew W. Appel, Christopher W. Fraser, David R. Hanson, and Arthur H. Watson, ‘‘Generating code for the Case statement,’’ *in preparation*.
6. V. Vyssotsky and P. Wegner, *A graph theoretical Fortran source language analyzer,* AT&T Bell Laboratories, Murray

Hill, NJ, 1963.

7. G. Cousineau, P. L. Curien, and M. Mauny, ‘‘The Categorical Abstract Machine,’’ in *Functional Programming Languages and Computer Architecture, LNCS Vol 201*, ed. J. P. Jouannaud, pp. 50-64, Springer-Verlag, 1985.
8. Andrew W. Appel and Trevor Jim, ‘‘Optimizing closure environment representations,’’ CS-TR-168-88, Princeton University, 1988.
9. David Ungar, ‘‘Generation scavenging: a non-disruptive high performance storage reclamation algorithm,’’ *SIGPLAN Notices (Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments)*, vol. 19, no. 5, pp. 157-167, ACM, 1984.
10. A. W. Appel, ‘‘Garbage collection can be faster than stack allocation,’’ *Information Processing Letters*, vol. 25, no. 4, pp. 275-279, 1987.
11. David R. Chase, ‘‘Safety considerations for storage allocation optimizations,’’ *SIGPLAN ’88 Conf. on Prog. Lang. Design and Implementation*, pp. 1-10, ACM, 1988.