Space-Efficient Closure Representations*

Zhong Shao and Andrew W. Appel †

Princeton University CS-TR-454-94
March 1994

Abstract

Many modern compilers implement function calls (or returns) in two steps: first, a *closure* environment is properly installed to provide access for free variables in the target program fragment; second, the control is transferred to the target by a "jump with arguments (or results)." *Closure conversion*, which decides where and how to represent closures at runtime, is a crucial step in compilation of functional languages. We have a new algorithm that exploits the use of compile-time control and data flow information to optimize closure representations. By extensive closure sharing and allocating as many closures in registers as possible, our new closure conversion algorithm reduces heap allocation by 36% and memory fetches for local/global variables by 43%; and improves the already-efficient code generated by the Standard ML of New Jersey compiler by about 17% on a DECstation 5000. Moreover, unlike most other approaches, our new closure allocation scheme satisfies the strong "safe for space complexity" rule, thus achieving good asymptotic space usage.

1 Introduction

Many modern compilers take great efforts to optimize function calls and returns because they are fundamental control structures, especially in functional languages. Before a function call, context information is saved from registers into a "frame." In a compiler based on Continuation-Passing Style (CPS), this "frame" is the closure of a continuation function [33].

In a CPS-based compiler, a *closure* environment is constructed at each function (or continuation) definition site; it provides runtime access to bindings of variables free in the function (or continuation) body. Each function call is then implemented as first installing the corresponding closure environment, setting up the arguments (normally in registers), and then passing the control to the target by a "jump" instruction. Function returns are implemented in the same way because they are essentially calls to continuation functions, if represented in CPS.

A closure can be any combination of registers and memory data structures that gives access to the free variables [25, 6]. The compiler is free to choose a closure representation that minimizes stores (closure creation), fetches (to access free variables), and memory use (reachable data).

We have developed a new algorithm for choosing good closure representations. As far as we know, our new closure allocation scheme is the first to satisfy all of the following important properties:

- Unlike stack allocation and traditional linked closures, our shared closure representations are safe for space complexity (see Section 2); at the same time, they still allow extensive closure sharing.
- Our closure allocation scheme exploits extensive use of compile-time control and data flow information to determine the closure representations.
- Source-language functions that make several sequential function calls can build one shared closure for use by all the continuations, taking advantage of callee-save registers.
- Because activation records (i.e., frames) are also allocated in the heap, they can be freely shared with other heap-allocated closures. Under stack allocation, this is impossible since stack frames normally have shorter lifetime than heap-allocated closures.
- Tail recursive calls—which are often quite troublesome to implement correctly on a stack [20]—can be implemented very easily.
- All of our closure optimizations can be cleanly represented using continuation-passing and closure-passing style [4] as the intermediate language.
- Once a closure is created, no later writes are made to it; this makes generational garbage collection and *call/cc* efficient, and also reduces the need for alias analysis in the compiler.

*To appear in *ACM Conference on Lisp and Functional Programming*, June 1994

† {zsh,appel}@cs.princeton.edu. Supported in part by the National Science Foundation Grant CCR-9002786 and CCR-9200790.

- Because all closures are either allocated in the heap or in registers, first class continuations *call/cc* are very efficient, requiring no complicated stack hackery [21].

Our new closure allocation scheme does not use any runtime stack. Instead, all closure environments are either allocated in the heap or in registers. This decision may seem controversial, because stack allocation is widely believed to have better reference locality, and deallocation of stack frames can be cheaper than garbage collection. Moreover, because heap allocated closures are not contiguous in memory, an extra memory write and read (of the frame pointer) are necessary at each function call. These assumptions no longer hold, for three reasons:

1. As we will show in Section 4, because most parts of continuation closures are allocated in callee-save registers [6], the extra memory write and read at each call can often be avoided. With the help of compile-time control and data flow information, the combination of shared closures and callee-save registers can often be comparable to or even better than stack allocation [7].
2. In a companion paper [7], we show that stacks do not have a significantly better locality of reference than heap-allocated activation records, *even in a modern cache memory hierarchy.* Stacks do have a much better *write miss ratio*, but not a much better *read miss ratio*. But on many modern machines, the *write miss penalty* is approximately zero [23, 16, 7].
3. The amortized cost of collection can be very low [1, 7], especially with modern generational garbage collection techniques [36].

The major contribution of our paper is a "safe for space" *closure conversion* algorithm that integrates and improves most previous closure analysis techniques [26, 6, 33, 30, 20, 22], using a simple and general framework expressed in continuation-passing and closure-passing style [4, 6]. Our new algorithm extensively exploits the use of compile-time control and data flow information to optimize closure allocation strategies and representations. Our measurements show that the new algorithm reduces heap allocation by 36% and memory fetches for local/global variables by 43%; and improves the already-efficient code generated by the Standard ML of New Jersey compiler by about 17% on a DECstation 5000.

2 Safely Linked Closures

Optimization of closure representations is sometimes dangerous and unsafe for space usage (i.e., maximum live data size). In 1988, Chase [11] observed that certain storage allocation optimization may convert a program that runs robustly into one that does not, due to the requirement of larger fraction of memory than the program actually needs. Appel [2] also noticed that programs using linked closures[1], or stack-allocated activation records, may cause a compiled program to use much more memory.

```
fun f(v,w,x,y,z) =
  let fun g() =
        let val u = hd(v)
            fun h() =
              let fun i() = w+x+y+z+3
               in (i,u)
              end
         in h
        end
   in g
  end

fun big n = if n<1 then [0] else n :: big(n-1)

fun loop (n,res) =
  if n<1 then res
  else (let val s = f(big(N),0,0,0,0)()
         in loop(n-1,s::res)
        end)

val result = loop(N,[])
```

Figure 2: An Example in Standard ML

For example, consider the Standard ML [29] program in Figure 2. With flat closures[2] (see Figure 1), each evaluation of `f(...)()` yields a closure `s` for `h` that contains just a few integers `u`, `w`, `x`, `y`, and `z`; the final result (e.g., `result`) contains N copies of the closure `s` for `h`, thus it uses at most $O(N)$ space. With linked closures (see again Figure 1), each closure `s` for `h` contains a pointer to the closure for `g`, which contains a list `v` of size N. Since the final result keeps N closures for `h` simultaneously, it requires $O(N^2)$ space consumption instead of $O(N)$. Obviously, this space leak is caused by inappropriately retaining some "dead" objects (`v`) that should be garbage collected earlier.

In 1992, we found several instances of real programs whose live data size (and therefore memory use) was unnecessarily large (with factors of 2 to 80) when compiled by early versions of our compiler that introduced this kind of space leak. All recent versions of SML/NJ

[1] A *linked closure* [27] is a record that contains the bound variables of the enclosing function, together with a pointer to the enclosing function's closure.

[2] A *flat closure* [10] is a record that holds only the free variables needed by the function.

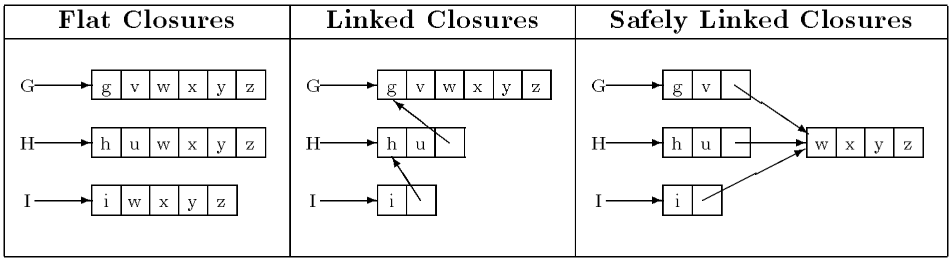


Figure 1: A comparison of three closure representations

V	::=	*variable*	D	::=	D_1 D_2 | `fun` F | `val` V = `select`(I,A)
I	::=	*integer constant*		|	`val` $V = (A_1, A_2, \ldots, A_n)$
R	::=	*real constant*		|	`val` $V = P(A_1, A_2, \ldots, A_n)$
P	::=	*arithmetic operator*			
A	::=	$V \mid I \mid R$	E	::=	`if` V `then` E_1 `else` E_2
F	::=	$V_0(V_1, V_2, \ldots, V_n) = E$ | F_1 `and` F_2		|	`let` D `in` E_1 `end` | $A_0(A_1, A_2, \ldots, A_n)$

Figure 3: Abstract syntax of CPS

have obeyed the "safe for space complexity" (SSC) rule, and users really did notice the improvement. The SSC rule is stated as follows:*any local variable binding must be unreachable after its last use within its scope* (see Appel [2] for a more formal definition).

Traditional stack allocation schemes and linked closures obviously violate this rule because local variable bindings will stay on the stack until they exit their scope, so may remain live even after their last use. Flat closures do satisfy the SSC rule, but they require that variables be copied many times from one closure to another. Many of the closure strategies described by Appel and Jim [3] violate the rule.

Most stack-frame implementations also violate SSC, since dead variables remain in the frame until a function returns. This can be avoided by associating a descriptor with each return address, showing which variables are live; but this complicates the garbage collector [7, 8].

Obeying SSC can require extra copying of pointer values from an old closure that contains them (but also contains values not needed in a new context) into a new closure. One cannot simply "zap" the unneeded values in the old closure, since it is not known whether there are other references to the old closure. The challenge is to find efficient closure strategies that obey SSC while minimizing copying.

Our new algorithm uses *safely linked closures* (the 3rd column in Figure 1), which contain only those variables actually needed in the function[3], but avoids closure copying by grouping variables with same *lifetime* into a sharable record.

In Figure 1, we use G, H and I to denote the closure, and g, h, and i for code pointers. With flat closures, variables `w`, `x`, `y`, and `z` must be copied from the closure of `g` into the closure of `h`, and then into the closure of `i`, this is very expensive. With traditional linked closures, closures for `h` and `i` are unsafely re-using the closure for `g`, retaining the variable `v` that is not free in `h` or `i`; moreover, accessing variables `w`, `x`, `y` and `z` inside I is quite expensive because at least two links needs to be traversed. By noticing that `w`, `x`, `y`, and `z` have same lifetime, the *safely linked closure* for `g` puts them into a separate record, which is later shared by closures for `h` and `i`. Unlike linked closures, the nesting level of safely linked closures never exceeds more than two, so they still enjoy very fast variable access time.

3 Continuations and Closures

We will illustrate CPS-conversion (which is not new [33, 26, 2]), and our new closure analysis algorithm, on the example in Figure 4. The function `iter` iteratively applies function f to argument x until it converges to satisfy predicate p.

[3] In practice, both this and SSC can be relaxed a little because the asymptotic space complexity will not change if we retain some variables that can be proven of constant size at compile time.

```
fun iter(x,p,f) =
  let fun h(a,r) = if p(a,r) then a
                   else h(f(a),a)
   in h(x,1.0)
  end
```

Figure 4: Function iter in Standard ML

3.1 Continuation-passing style

Continuation-passing style (CPS) is a subset of λ-calculus, but which closely reflects the control-flow and data-flow operations of a von Neumann machine. As in λ-calculus, functions are nested and variables have lexical scope; but as on a von Neumann machine, order of evaluation is pre-determined. For the purposes of this paper, we express CPS using ML notation, albeit severely constrained — see Figure 3. An atom A can be a variable or a constant; a *record* can be constructed out of a sequence $(A_1, A_2, \ldots, A_n)$ of atoms. If v is bound to an n-element record, then the i^{th} field may be fetched using `select`(i,v); The syntax for building records, selecting fields, applying primitive arithmetic operators, and defining mutually recursive functions (`fun` and F) must specify a continuation expression E that will use the result (via `let` expressions). [4] On the other hand, function application (shown in the last line on the right of Figure 3) does not specify a continuation expression — functions never *return* in the conventional sense. Instead, it is expected that many functions will pass a *continuation function* as one of their arguments. This function can be defined in the ordinary way (by `fun`), and will presumably be invoked by the callee in order to continue the computation.

```
fun iter(C,x,p,f) =
  let fun h(a,r) =
        let fun J(z) = if z then C(a)
                       else (let fun Q(b) = h(b,a)
                              in f(Q,a)
                             end)
         in p(J,a,r)
        end
   in h(x,1.0)
  end
```

Figure 5: Function iter after CPS-based optimizations

Figure 5 shows the code of the function `iter` after translation into CPS, and after the continuation argument of `h` has been hoisted out of the loop because it is loop-invariant [32]. Such optimizations are performed after CPS-conversion, but before the closure analysis that is the subject of this paper.

To ease the presentation, we use capital letters to denote continuations (e.g., `C`, `J`, and `Q`). We call those functions declared in the source program *user functions* (e.g., `iter`, `h`), and those introduced by CPS conversion*continuation functions* (e.g., `J`, `Q`). *Continuation variables* are all those formal parameters (commonly placed as the first argument) introduced in CPS conversion to serve as return continuations (e.g., `C`). Functions such as `iter`, `p` and `f` are called *escaping functions*, because they may be passed as arguments or stored in data structures so that the compiler cannot identify all the places where they are called. All functions that do not escape are called *known functions* (e.g., `h`). We can do extensive optimizations on known functions since we know all of their call sites at compile time.

3.2 Closure-passing style

Continuation-passing style is meant to approximate the operation of a von Neumann computer; a "function" in machine language is just an address in the executable program, perhaps with some convention about which registers hold the parameters—very much like a "jump with arguments." The notion of function in CPS is almost the same, except that they have nested lexical scope and may contain *free variables*. This problem is solved by adding a *closure* which makes explicit the access to all nonlocal variables.

Kranz [25, 26] showed that different kinds of functions should use different closure allocation strategies. For example, the closure for a *known function* (e.g., `h` in Figure 5) can be allocated in registers, because we know all of its call sites at compile time and can let the caller always pass its free variables as extra arguments at runtime; on the other hand, the closure for an *escaping function* may have to be allocated as a heap record that contains both the machine code address of the function plus bindings for all its free variables.

Conventional compilers use *caller-save* registers, which may be destroyed by a procedure call, and *callee-save* registers, which are preserved across calls. Variables not live after the call may be allocated to *caller-save* registers which cuts down on register-saving traffic.

We wanted to adapt this idea to our continuation-passing intermediate representation. We did so as follows [6]: each CPS-converted user function f is passed its ordinary arguments, a continuation function c_0, and k extra arguments $c_1, \ldots, c_k$. The function "returns" by invoking c_0 with a "result" argument r and the additional arguments $c_1, \ldots, c_k$. Thus, the "callee-save" arguments $c_1, \ldots, c_k$ are handed back to the continuation. When this CPS code is translated into machine instructions, $c_1, \ldots, c_k$ will stay in registers throughout the ex-

[4] Later in the paper, we use `let` E_1 E_2 ... E_n `in` ... `end.` to denote a sequence of `let` expressions, e.g., `let` E_1 `in` (`let` E_2 `in` ... (`let` E_n `in` ... `end`) ... `end`) `end`.

```
01    fun iter(I,C0,C1,C2,C3,x,p,f) =
02      let fun h(a,r,CR,p) =
03            let fun J0(J1,J2,J3,z) =
04                   if z then
05                     (let val C0 = select(0,J1)
06                          val C1 = select(1,J1)
07                          val C2 = select(2,J1)
08                          val C3 = select(3,J1)
09                       in C0(C1,C2,C3,J2)
10                      end)
11                   else
12                     (let fun Q0(Q1,Q2,Q3,b)
13                                = h(b,Q2,Q1,Q3)
14                          val f = select(4,Q1)
15                          val f0 = select(0,f)
16                       in f0(f,Q0,J1,J2,J3,J2)
17                      end)
18                 val p0 = select(0,p)
19              in p0(p,J0,CR,a,p,a,r)
20             end
21          val CR = (C0,C1,C2,C3,f)
22       in h(x,1.0,CR,p)
23      end
```

Figure 6: Function `iter` in after closure conversion

ecution of f; unless f needed to use those registers for other purposes, in which case f must save and restore them. One could also say that the continuation is represented in $k+1$ registers ($c_0, ..., c_k$) instead of in just one pointer to a memory-resident closure.

In our previous work [6], we outlined this framework and demonstrated that it could reduce allocation and memory traffic. However, we did not have a really good algorithm to exploit the flexibility that callee-save registers provide.

Closure creation and use can also be represented using the CPS language itself [4, 24]. We call this *closure-passing style* (CLO). The main difference between CLO and CPS is that functions in CLO do not contain free variables, so they can be translated directly into machine code. In CLO, the formal parameters of each function correspond to the target machine registers, and heap-allocated closures are represented as CPS records.

Figure 6 lists the code of function `iter` after translation into CLO. All continuation functions and variables (e.g., `C,J,Q`) are now represented as a machine code pointer (e.g., `C0,J0,Q0`) plus three extra callee-save arguments (e.g., `C1-C3,J1-J3,Q1-Q3`).

The original function `J` (in Figure 5) had free variables `C,f,a,h`. With three callee-save registers, `C` becomes the four variables `C0,C1,C2,C3`, for an effective total of seven. When `J` is passed to `p` (line 19), these seven free variables—plus the machine code pointer for `J`'s entry point—must be squeezed into four formal parameters `J0,J1,J2,J3`. Where there are more than three free variables, some of the callee-save arguments must be heap-allocated records containing several variables each; thus, the `CR` closure-record appears as `J1` in the call on line 19.

Previous closure conversion algorithms [33, 25, 4] require memory stores for each continuation function. An important advance in our new work is that we allocate (in this example) only one record `CR` for the functions `J,Q,h`, *and this record is carefully chosen to contain loop-invariant components, so that it can be built outside the loop.*

Escaping user functions (`iter,p,f`) are now represented as a closure record (`I,p,f`), each with its 0^{th} field being the machine code pointer (`iter,p0,f0`). Escaping function calls are implemented as first selecting the 0^{th} field, placing the closure itself in a special register (the first formal parameter), and then doing a "jump with arguments" (lines 15-16, 18-19).

4 Closure Conversion

In this section, we present our new *closure conversion* algorithm using the framework defined in Section 3. Our algorithm takes a CPS expression E as the argument, determines the closure representation for each function definition in E, and then converts E into a CLO expression E' in which each function definition does not contain any free variables. The presentation of our algorithm is organized in the following five steps:

1. Construct an *extended* CPS *call graph* that captures the control flow information in the CPS expression.

2. Gather the set of *raw free variables* and their lifetime information for each CPS function.

3. Use *closure strategy analysis* to determine where in the machine to allocate each closure.

4. Use *closure representation analysis* to determine the actual structure of each closure at runtime.

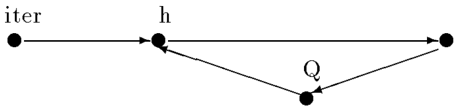
5. Find out the variable access path for all non-local variables of each CPS function.

Each step here does not necessarily correspond to a separate pass in the real implementation because many of them are actually done in a single pass.

4.1 Extended CPS call graph

Given a CPS expression E, we can divide the set of function definitions in E into four categories: *escaping user functions*, *known user functions*, *escaping continuation functions*, and *known continuation functions* (see the last paragraph of Section 3.1 for definitions). Given two CPS variables v and w, v *directly calls* w if w is possibly the first function call inside the function definition of v. For example, in Figure 5, `J` directly calls `C` and `f` but not `h`, because `h` cannot be the first call inside `J`.

The *extended* CPS *call graph* G of E is a directed graph with the set of function definition variables in E as nodes; there is an edge from v to w in G if v directly calls w, or v directly calls some function with w as its return continuation. For example, the *extended* CPS *call graph* for the function `iter` in Figure 5 is as follows:



Although `J` is not directly called by `h`, we conservatively assume that the function `p` will always call its return continuation, i.e., `J`.

The extended CPS call graph G of E essentially captures a very simple set[5] of control flow information in E. Cycles in the graph imply loops or recursions (e.g., the path from `h` to `J` to `Q`). The nested hierarchies of loops and recursions in E can be revealed by running the Tarjan interval analysis[6] algorithm [34, 31] on G, assuming G is a reducible flow graph. For the purpose of our closure analysis, this control flow information can be used to choose closure representations that allow more efficient variable accesses in frequently-executed program fragments (e.g., loops).

For every function definition v in E, we define its *loop level* $L(v)$ as the nesting depth of its interval in the extended CPS call graph for E, assuming the outmost interval is at depth 0. For all other function variables that are not actually defined in E (e.g., `C,f,p` in Figure 5), their *loop levels* are defined as 0. The *loop level* of each call from v to w is defined as $L(v,w) = min(L(v), L(w))$. The *loop level* for an arbitrary CPS expression inside a function definition v is inductively defined as follows:

- $L($`if` V `then` E_1 `else` $E_2) = max(L(E_1), L(E_2))$;
- $L($`let` D `in` E_1 `end`$) = L(E_1)$;
- $L(A_0(A_1,A_2,\ldots,A_n)) = L(v, A_0)$ if A_0 is a continuation, and $max(L(v, A_0), L(v, A_1))$ if A_0 is a user function and A_1 is its return continuation.

The loop level number can be used as a guide for static branch prediction of control flow in E. For example, in function `iter`, the loop level of `h,J,Q` is 1, and the loop level of `iter,C,f,p` is 0. In the CPS expression "`if z then ... else ...`" in `J`'s definition, `J` either calls `f` with the return continuation `Q`, or it calls the continuation variable `C`. Clearly, the call to `f` and `Q` is inside a loop because $L($`J`$,$`Q`$) = 1$, while the call to `C` is not, because $L($`J`$,$`C`$) = 0$. The closure representations for `J` and `h` should take more considerations on the "`else`" branch because it is more likely to be taken at runtime.

For each function definition w in the expression E, we also define *pred*(w) as its predecessor set, i.e., the set of all variables v such that there is an edge from v to w in E's extended CPS call graph.

4.2 Raw free variables with lifetime

To implement the *safely-linked closures* described in Section 2, we want to group variables into closure records if they have similar lifetimes. If v is defined much later than w, x, y, then we may not have enough registers to hold w, x, y while waiting for v. If y's last use is much earlier than w's or x's, then the record (w, x, y) might not obey the SSC rule.

Most closure conversion algorithms [2, 26, 33] start with a phase to gather the set of *raw* free variables for each function definition in E. These free variables are called *raw free variables* because some of them may be substituted by a set of other free variables later during the *closure representation analysis* phase; we use the *true free variables* to denote the set of variables that are finally put in the closure environment.

Our algorithm does the same except that we are also gathering the *lifetime* information at the same time. To define the lifetime for a variable, we first assign a *stage number* (denoted as SN) for each function definition w using the following method:

- if w is the outmost function definition, then $SN(w) = 1$;
- if w is a user function, then $SN(w) = 1 + SN(f)$ where f is the nearest enclosing function definition;
- if w is a continuation function, then $SN(w) = 1 + max\{\ SN(v) \mid v \in pred(w)\ \}$ (this definition is valid because continuation functions are never recursive).

We then define the *use time* for each use of every CPS variable v as $SN(f)$ where f is the nearest enclosing function definition for this use of v. The set of free variables for each function definition f is now a set of triples (v, fut, lut) where v is the variable, *fut* is the *first use time* of v denoting the smallest stage number of all uses of v inside f, and *lut* is the *last use time* of v denoting the largest stage number of all uses of v inside f.

To reflect the control flow, the *lut* and *fut* numbers of v can also be calculated based on the (predicted)

[5] Shivers [32] presented more sophisticated techniques that can find even better approximations of control flow information.

[6] Given a flow graph G, a Tarjan interval is essentially a single-entry, strongly-connected subgraph of G; the interval analysis [31] partitions the set of nodes in G into disjoint intervals, with each interval representing a proper loop (or recursion) layer.

Function	Stage Number	Raw Free Variables	Closure Strategy
`iter`	1	$\emptyset$	1 slot
`h`	2	$\{(\mathtt{p},2,2),(\mathtt{C},3,3),(\mathtt{f},3,3),(\mathtt{h},4,4)\}$	2 slots
`J`	3	$\{(\mathtt{C},3,3),(\mathtt{f},3,3),(\mathtt{a},3,4),(\mathtt{h},4,4)\}$	3 slots
`Q`	4	$\{(\mathtt{a},4,4),(\mathtt{h},4,4)\}$	3 slots

Table 1: Raw free variables and closure strategies

execution frequency of each use of v. For example, for a CPS expression **if** V **then** E_1 **else** E_2, we can ignore all uses of v in E_1 (or E_2) if $L(E_1) > L(E_2)$ (or $L(E_2) > L(E_1)$) during the calculation. The higher preference for those uses inside a loop body would likely lead to more efficient closure representation at runtime.

For example, the stage number and the set of raw free variables for all function definitions in Figure 5 are shown in Table 1. Notice that a variable can have different *lut* and *fut* numbers inside different function definitions (e.g., `a` in `J` and `Q`).

4.3 Closure strategy analysis

Closure strategy analysis essentially determines where in the machine to allocate each closure. Unlike previous CPS compilers [26, 33], we do not do any *escape analysis*[7] because we simply do not use a runtime stack. Our *closure strategy analysis* only decides how many *slots* (i.e., registers) each closure is going to use, denoted by $S(f)$ for each function f. We calculate $S(f)$ using the following simple algorithm:

- If f is an escaping user function, then $S(f) = 1$. This essentially means that all its free variables must be put in the heap. The closure for f is a pointer to a linked data structure in the heap.
- If f is an escaping continuation function, then $S(f) = k$ where k is the number of callee-save registers. Because their call sites are not known at compile time, most continuation functions have to use the uniform convention, i.e., always in k callee-save registers [6]. In special cases, some continuation functions can be represented differently; this will be briefly discussed in Section 5.3.
- For known functions, since their call sites are all known at compile time, their closures (or environments) may be allocated completely in registers. However, the number of registers on the target machine can be limited, and it may not always be desirable to allocate all free variables in registers (see Section 5.2). We run the following iterative algorithm to calculate the appropriate number of slots (registers) used for each known function:
  1. Initially, each known function f is assigned m slots, i.e., $S(f) = m$, where m is the maximum number of available registers on the target machine minus the number of formal parameters of function f (assuming they will be passed in registers);
  2. Then, for each known function f, we substitute $S(f)$ by $min(\{T(v_1), \ldots, T(v_n), S(f)\})$. Here $v_1, ..., v_n$ are any subset of the functions in $pred(f)$ that do not enclose f's definition, i.e., f must be free in these $v_1, ..., v_n$. The value $T(v)$ is $max(1, S(v) - j)$ where j is the number of variables that are free in v but not in f. This substitution process is then repeated until $S(f)$ no longer changes and a fixed point[8] is reached.

  The second step here is based on the observation that if f is called inside a function v, and f is also free in v, then the number of slots assigned to f should not be bigger than the number of slots available for v's environment, otherwise, some kind of spilling will be inevitable.

  When choosing which subset of v_i to use in calculating $S(f)$ at step 2, we can again take advantage of the control flow information in the extended CPS call graph. More specifically, we want to favor those program fragments that are likely executed more often than others, so we always choose those v_i which has a higher $L(v_i, f)$ value (i.e., the call from v_i to f is within a loop).

Let's apply this algorithm to the function `iter` in Figure 5. Suppose we use 3 callee-save registers, then both $S(\mathtt{Q})$ and $S(\mathtt{J})$ are 3; $S(\mathtt{h})$ is initially 14, assuming that there are 16 available registers on the target machine; then since `Q` calls `h`, and `a` is free in `Q` but not in `h`, so $S(\mathtt{h})$ should be $min(3-1, 14)$, which is 2, as shown in Table 1; notice that the call from `iter` to `h` is not considered here because `h` is not free in `iter`.

4.4 Closure representation analysis

Closure representation analysis solves the following problem: "Given a function f, if f contains m free variables

[7] The *escape analysis* here refers to the analysis that decides whether a function's environment can be allocated on the stack or not.

[8] This iterative process clearly terminates because $T(v) \geq 1$, $S(f) \geq 1$, and the sum of $S(f)$ (for all functions) gets smaller in each round.

and is assigned n slots, how to place these m values into n slots?"

Given a CPS expression E, the closure representation analysis is done by processing each function definition through a preorder traversal of E; during the traversal, we maintain and update the following three data structures:

whatMap A static environment that maps every function definition processed so far to its closure representation.

whereMap A list of currently visible closures and variables.

baseRegs The current contents of callee-save registers.

When traversing and processing each function definition f, we do the following:

1. Suppose the set of *raw free variables* of f found in the last step (i.e., Section 4.2) is RFV, we first check if f is recursive or mutually recursive with some other functions, and then find the transitive closure RFV^* of f's raw free variables. For example, as shown in Table 1, function `h` is recursive and its RFV is $\{(\texttt{p},2,2),(\texttt{C},3,3),(\texttt{f},3,3),(\texttt{h},4,4)\}$; we remove `h` and replace it by its raw free variables. We also propagate `h`'s *fut* and *lut* numbers into each of its free variables by taking the minimum of their *fut* numbers and the maximum of their *lut* numbers. As the result, the transitive closure RFV^* of h is $\{(\texttt{p},2,4),(\texttt{C},3,4),(\texttt{f},3,4)\}$.

2. Next, we find the set of *true free variables* TFV of f by replacing each continuation variable in RFV^* by its corresponding callee-save variables, and each function definition by its closure contents (or slot variables). For example, suppose we use three callee-save registers, each continuation variable `C` is then represented by a code pointer `C0` and its three callee-save variables `C1,C2,C3`. The set of true free variables TFV for `h` is $\{(\texttt{p},2,4)$, $(\texttt{C0},3,4)$, $(\texttt{C1},3,4)$, $(\texttt{C2},3,4)$, $(\texttt{C3},3,4)$, $(\texttt{f},3,4)\}$. Notice that `C0,C1,C2,C3` here naturally inherit `C`'s *fut* and *lut* numbers.

3. Now assume that TFV of f contains m variables, and f is assigned n slots by closure strategy analysis in Section 4.3. If $m \leq n$, then we are done; otherwise, we search through the current list of visible closures maintained in the **whereMap** data structure, and see if there is any closure record that we can reuse (or share). The SSC rule mentioned in Section 1 is satisfied by making sure that we only reuse those closures whose contents are a subset of TFV. Because all closures in the heap are *safely linked closures*, certain closure sharings had already been anticipated while processing the enclosing function definitions. If there are multiple sharable closures, we use a "best fit" heuristic to decide which one to reuse. In the example of function `iter`, the closure `CR` (line 21 in Figure 6) is sharable by the continuations `J` and `Q`.

4. If the size m of TFV after closure sharing is still larger than n, we have to heap allocate part of the closure. We do this by putting $n-1$ variables into one slot each, and packing the remaining $m-n+1$ variables into the heap closure. The criteria in choosing these $n-1$ variables is based on the following priorities: the first priority is smaller *lut* number; the second is smaller *fut* number; the third is whether the variable is already in the current callee-save registers (i.e., **baseRegs**) or not. We also use the contents of **baseRegs** to decide which variable goes to which slot to avoid any possible register moves. For example, the function `h` is assigned 2 slots but `h` has 6 true free variables, we put the free variable `p` in the register because it has the smallest *fut* number (all variables have the same *lut* number).

5. Finally, we decide the actual layout of the spilled heap closure of the above $m-n+1$ variables based on each variable's *lut* number. To satisfy SSC with shared closures, each distinct *lut* number requires a separate record. For example, the closure for G in Figure 1 was split into two records because `v`'s *lut* number was different from those of `w,x,y,z`.

We finish processing the function definition f by updating the **whatMap**, **whereMap** and **baseRegs** environments accordingly based on f's closure representation.

Not only is `CR` shared in Figure 6, but its creation is outside the `h` loop. Thus, each iteration of `h` manages to call two unknown (escaping) functions *without any memory traffic!* This is one of the most important strengths of our new algorithm.

4.5 Access path for non-local free variables

Finding out the access path for each non-local free variable v is just a breadth-first search of v in the **whereMap** environment. We use the "lazy display" technique used by Kranz [26], so that loads of common paths can be shared. More specifically, let's look at the function `i` (the innermost function inside `f`) in Figure 2: assuming that `i` uses the safely linked closure shown in Figure 1, then accessing each non-local variable (e.g., `w,x,y,z`) inside `i` requires traversing two links; but we can first load the 2nd field of the closure `I` into a register r, and then access `w`, `x`, `y`, and `z` directly from r via one load. These intermediate variables (e.g., register r) may use up all the available machine registers and cause unnecessary register spilling, but this can always be avoided by selectively keeping limited number of intermediate variables in the "lazy display" (registers).

4.6 Remarks

Graph-coloring global register allocation and targeting, which have been implemented by Lal George [17], will accomplish most control transfers (function calls) (such as line 12 and 13 in Figure 6) without any register-register moves. This allows a more flexible boundary between callee-save and caller-save registers than is normal in most compilers.

Programs, in our scheme, tend to accumulate values in registers and only dump them into a closure at infrequent intervals. It may be useful to use more callee-save (and fewer caller-save) registers to optimize this.

Our closure scheme handles tail calls very nicely, simply by re-arranging registers. Hanson [20] shows how complicated things become when it's necessary to re-arrange a stack frame.

A source-language function that calls several other functions in sequence would, in previous CPS compilers (including our own) allocate a continuation closure for each call. The callee-save registers and safely linked closures allow us to allocate only once.

General deep recursions are handled very efficiently in our scheme. A conventional stack implementation tends to have a high space overhead per frame, but our closures are quite concise. Thus, total memory usage (and cache coverage) of recursions will be much less.

5 Case Studies

A good environment allocation scheme must implement frequently used control structures very efficiently. Many compilers identify special control structures at compile time, and assign each of them a special closure allocation strategy. For example, in Kranz's Orbit compiler [26], all tail recursions are asssigned a so-called "*stack/loop*" strategy, and all general recursions are assigned a "*stack/recursion*" strategy. Our new closure conversion algorithm, on the other hand, uniformly decides the closure strategy (i.e., number of slots) and the closure representation for each function solely based on the lifetime information of its free variables and simple control flow information.

In Section 3, we have shown how our new algorithm implements tail recursion very efficiently (i.e., function `iter`). In this section, we use several more examples to show how our new algorithm effectively deals with other common control structures such as a sequence of function applications, calling a known function, and general recursion.

5.1 Function calls in sequence

One common control structure in functional programs is making a sequence of function applications, as shown in the following example:

```
fun f(g,u,v,w) =
  let val x = g(u,v)
      val y = g(x,w)
      val z = g(y,x)
   in x+y+z+v+1
  end
```

Here the function `g` (a formal parameter of `f`) is called three times in a row inside the function `f`. Under the traditional stack scheme, when function `f` is called, an activation record for `f`—containing formal parameters (i.e., `g,u,v,w`) and local variables (i.e., `x,y,z`)—will be pushed onto the stack. Each time before `g` is called, certain local variables in registers must be saved onto the stack. For example, assuming all function arguments (i.e., `g,u,v,w`) and return results (i.e., `x,y,z`) are passed in registers, then before the first call to `g`, the registers holding `g` and `w` must be saved so that they can still be retrieved later after `g` returns.

If activation records are allocated on the heap, things get much worse. Every time registers need to be saved before a function call, a closure record has to be built on the heap. Because heap allocated closures are not contiguous in memory, an extra memory write (and later a memory read) of the frame pointer is necessary at each function call.

With our new closure analysis technique to make good use of callee-save registers, heap-allocated activation records can be made almost as efficient as stack allocation [7]. The idea is that we can always allocate most parts of the current activation record in callee-save registers. With careful lifetime analysis, register save/restore around several function calls can often be eliminated or amalgamated, so function calls in sequence need just write one heap record. Figure 7 lists the code of function `f` after translation into CLO (by our new algorithm in Section 4). Continuations are still represented as one code pointer plus three callee-save registers, all denoted by capital letters. As before, escaping function calls (i.e., calls to `g` on line 14,17,21) are implemented as first selecting the 0^{th} field, placing the closure itself in a special register (the first formal parameter), and then doing a "jump with arguments" (lines 13-14,16-17,20-21). Before the first call to `g` (line 21), we put variables that have smaller *lut* numbers (i.e., `g,w`) callee-save registers (i.e., `J2,J3`), and spill the rest (i.e., `C0-C3,v`) into a heap record `CR` (line 19). At the second and the third calls to `g` (line 17,14), no register save/restore are necessary. This is because the lifetime of `w` and `x` (also `g` and `y`) does not overlap, so they can just share one callee-save register (i.e., `J3` and `K3`, `K2` and `Q2`).

```
fun f(C0,C1,C2,C3,g,u,v,w) =
 let fun J0(J1,J2,J3,x) =
       let fun K0(K1,K2,K3,y) =
             let fun Q0(Q1,Q2,Q3,z) =
                   let val v = select(4,Q1)
                       val r = Q3+Q2+z+v+1
                       val C0 = select(0,Q1)
                       val C1 = select(1,Q1)
                       val C2 = select(2,Q1)
                       val C3 = select(3,Q1)
                    in C0(C1,C2,C3,r)
                   end
                 val g0 = select(0,K2)
              in g0(K2,Q0,K1,y,K3,y,K2)
             end
           val g0 = select(0,J2)
        in g0(J2,K0,J1,J2,x,x,J3)
       end
     val CR = (C0,C1,C2,C3,v)
     val g0 = select(0,g)
  in g0(g,J0,CR,g,w,u,v)
 end
```

Figure 7: Making a sequence of function calls

5.2 Lambda lifting on known function

Lambda lifting [22] is a well-known transformation that rewrites a program into an equivalent one in which no function has free variables. Lambda lifting on known functions essentially corresponds to the special closure allocation strategy that allocates as many free variables in registers as possible. But this special strategy does not always generate very efficient code [26]. For example, in the following program, assume that `f` is a known function, and `p,w,x,y`, and `z` are its free variables.

```
fun f u = (p u, u+w+x+y+z+1)

fun g(x,y) = (p x, f x, f y)
```

If the closure for `f` is allocated in registers, then before the call to `p` inside `g`, some of `f`'s free variables must be saved onto the heap (assuming there are only three callee-save registers); when the call to `p` returns, these variables must be reloaded back into registers, and passed to function `f`; after entering `f`, some of them again have to be saved when `f` calls `p`, and so on. Clearly, allocating `f`'s environment in registers dramatically increases the need for more callee-save registers inside `g`. This leads to more memory traffic when there are only a limited number of callee-save registers.

The *closure strategy analysis* described in Section 4.3 uses an iterative algorithm to decide the number of registers assigned to each known function. The number of registers assigned to `f` will be restricted by those of its callers, i.e., the return continuation for `p x` and the return continuation for the first call to `f`. As a result, `f` is only assigned one slot, and its closure will be allocated in the heap.

5.3 General recursion

The *closure strategy analysis* algorithm described in Section 4.3 conservatively represents all continuation functions using the same (fixed) number of callee-save registers. This restriction can be relaxed: continuations that are passed to known functions can be represented in any number of callee-save registers. This special calling convention is especially desirable for general recursions such as the `map` function shown as follows (after translation into CPS):

```
fun map(C,f,l) =
  let fun m(J,z) =
        if (z=[]) then []
        else (let val a = car z
                  val r = cdr z
                  fun K(b) =
                    let fun Q(s) =
                          let val y = b::s
                           in J(y)
                          end
                     in m(Q,r)
                    end
               in f(K,a)
              end)
   in m(C,l)
  end
```

Notice that the recursive function `m` is called only at two places: one by function `map` with `C` as the return continuation, one inside `K` with `Q` as the return continuation. Because the second call to `m` is a recursive call, it will be executed much more often at runtime. We can represent all normal continuation functions in three callee-save registers, but represent continuations `J` and `Q` in two callee-save registers. Figure 8 lists the code of function `map` after translation into CLO using the above special calling convention.

Here `m` is a known function, and the environment for `m` (i.e., the free variable `f`) is allocated in a register (i.e., `f` is treated as an extra argument of `m`, see line 9,21,28). Since continuation `C` still uses the normal calling convention, when it is passed to the function `m` (line 28), a new "coercion" continuation (i.e., `R0` on line 2-7) has to be built to adjust the normal convention (three callee-save registers `C0-C3`) into the special convention (two callee-save registers `R0-R2`). Because the return continuation `J` of `m` is represented in two callee-save registers (i.e., `J0-J2`), we can build a smaller heap closure (of size 3, on line 23) for continuation `K`.

If both `J` and `Q` are represented in three callee-save registers, the heap closure for `K` would at least be of size 4.

Program	Size	Description
Barnes-Hut	3036	N-body problem solver.
Boyer	919	Standard theorem-prover benchmark.
CML-sieve	1356	CML implementation of prime number generator.
Knuth-Bendix	655	The Knuth-Bendix completion algorithm.
Lexgen	1185	A lexical-analyzer generator.
Life	148	The game of Life implemented using lists.
Ray	874	A simple ray tracer.
Simple	990	A spherical fluid-dynamics program.
VLIW	3658	A VLIW instruction scheduler.
YACC	7432	An implementation of an LALR parser generator.

Table 2: General Information about the Benchmark Programs

	Links Traversed (millions)			**Allocation Size** (megawords)			**Execution Time** (seconds)			**Code Size**
Program	**Old**	**New**	**Savings**	**Old**	**New**	**Savings**	**Old**	**New**	**Savings**	**Savings**
Barnes-Hut	54.70	33.35	39.03%	51.84	39.00	24.77%	33.11	28.14	15.01%	16.91%
Boyer	4.58	2.58	43.67%	11.23	5.78	48.53%	2.70	2.37	12.22%	13.36%
CML-sieve	44.11	25.43	42.35%	53.49	33.76	36.88%	35.02	30.55	12.76%	16.19%
Knuth-Bendix	16.09	13.27	17.53%	40.93	24.48	40.19%	8.02	6.25	22.07%	14.57%
Lexgen	14.98	7.57	49.47%	21.37	8.77	58.96%	12.33	11.07	10.22%	18.68%
Life	1.60	0.76	52.50%	2.36	1.61	31.78%	1.42	1.25	11.97%	14.83%
Ray	17.84	12.85	27.97%	28.32	25.96	8.33%	25.89	22.40	13.48%	19.65%
Simple	71.44	38.88	45.58%	70.39	38.31	45.57%	24.18	18.86	22.00%	38.63%
VLIW	39.99	16.90	57.74%	49.50	31.62	36.12%	20.81	12.05	42.10%	23.98%
YACC	10.97	4.86	55.70%	13.89	10.09	27.36%	4.94	4.35	11.94%	26.29%
Average			43.15%			35.85%			17.38%	20.31%

Table 3: Performance of the Benchmark Programs

	Escaping User (megawords)		**Known User** (megawords)		**Continuation** (megawords)		**Record** (megawords)		**Other** (megawords)	
Program	**Old**	**New**	**Old**	**New**	**Old**	**New**	**Old**	**New**	**Old**	**New**
Barnes-Hut	1.21	1.02	10.47	2.86	29.92	24.89	10.07	10.07	0.17	0.17
Boyer	1.91	1.18	0.00	0.48	4.88	3.17	0.95	0.95	3.49	0.00
CML-sieve	7.28	7.28	11.40	8.29	22.85	6.24	11.95	11.95	0.00	0.00
Knuth-Bendix	12.49	5.99	0.04	1.45	24.28	12.92	4.11	4.11	0.00	0.00
Lexgen	1.39	0.48	0.57	0.64	16.54	6.90	0.75	0.75	2.12	0.00
Life	0.06	0.06	0.00	0.06	1.44	0.63	0.79	0.79	0.08	0.08
Ray	0.00	0.00	0.01	2.11	14.42	9.96	13.89	13.89	0.00	0.00
Simple	7.37	5.87	3.01	1.30	54.10	25.23	5.92	5.92	0.00	0.00
VLIW	7.38	6.78	2.67	3.07	32.98	15.72	5.75	5.75	0.73	0.30
YACC	0.23	0.23	2.80	2.04	8.80	5.92	1.85	1.85	0.21	0.04

Table 4: Allocation Breakdown of the Benchmark Programs

```
fun map(C0,C1,C2,C3,f,l) =
  let fun R0(R1,R2,x) =
        let val C0 = select(0,R1)
            val C1 = select(1,R1)
            val C3 = select(2,R1)
         in C0(C1,R2,C3,x)
        end

      fun m(J0,J1,J2,z,f) =
        if (z=[]) then J0(J1,J2,[])
        else (let val a = car z
                  val r = cdr z
                  fun K0(K1,K2,K3,b) =
                    let fun Q0(Q1,Q2,s) =
                          let val y = Q2::s
                              val J0 = select(0,Q1)
                              val J1 = select(1,Q1)
                              val J2 = select(2,Q1)
                           in J0(J1,J2,y)
                          end
                     in m(Q0,K1,b,K2,K3)
                    end
                  val CR = (J0,J1,J2)
                  val f0 = select(0,f)
               in f0(f,K0,CR,r,f,a)
              end)
      val CC = (C0,C1,C3)
   in m(R0,CC,C2,l,f)
  end
```

Figure 8: Function `map` using special calling conventions

6 Measurements

We have implemented our new "space-efficient" closure conversion algorithm in the Standard ML of New Jersey compiler version 1.01. We compare the performance of two compilers, using our **Old** algorithm [2, 6] and the **New** algorithm described in this paper. The **Old** algorithm uses a hybrid scheme: it uses linked closure representation if it is space safe, otherwise it uses flat closure representation. Both the **Old** and **New** compilers satisfy the "safe for space complexity" rule. Both compilers represent continuation closures using three callee-save registers. Both compilers use *representation analysis* [28] to allow arguments being passed in registers. The "lazy display" technique is implemented in both compilers, however it is used more effectively in the **New** compiler because of its more extensive use of shared closures.

Table 2 shows the set of benchmarks we use and the source program size in number of lines. Table 3 shows the number of memory fetches for local/global variables, total heap allocation size, the execution time (user time plus system time) on a DECstation 5000, and the code size (only show the "savings") for both the **Old** and **New** compilers. On average, the **New** compiler reduces heap allocation by 36% and memory fetches for local/global variables by 43%; and improves the already efficient code generated by the **Old** compiler by 17%. The **New** compiler also uniformly generates more compact code, achieves an average of 20% reduction in code size over the **Old** compiler. The VLIW benchmark—an instruction scheduler—achieves up to 42% speedup in execution time, because it gets significant benefits from the extensive closure sharing in our new closure conversion algorithm.

We have also measured the allocation profile of various kinds of closures, shown in Table 4: **Escaping User**, **Known User**, and **Continuation** are respectively the total size of closures (in megawords) allocated for escaping user function, known user function, and continuation function; **Record** includes *cons* cells and other explicitly allocated non-closures; **Other** includes arrays, references, and register spills. Most of the reduction in heap allocation is from the continuation closures; closure analysis does nothing to reduce the allocation of records and arrays.

7 Comparison with Other Schemes

Our work on closure analysis is related to, and influenced by, many other research results.

Closure analysis: Steele [33] used continuation closures instead of "stack frames;" Rozas [30] and Kranz [25, 26] used closure analysis to choose specialized representations for different kinds of closures; Appel and Jim investigated closure-sharing strategies [3]. We combine all of these analyses (except stack allocation) and more.

Tail calls: Hanson [20] showed the complexity of implementing tail calls correctly and efficiently on a conventional stack.

Closure-passing style: Lambda notation has often been used to represent the results of closure analysis (this is also called "lambda lifting") [14, 22, 4, 24].

Efficient call/cc: Many have tried to make call/cc efficient [15, 13, 21].

Callee-save registers: Dataflow analysis can help decide whether to put variables in caller-save or callee-save registers [12, 19]. We had shown how to represent callee-save registers in continuation-passing style [6, 2] but our new algorithm does a much better job of it.

Safe space complexity: The notion that certain compiler optimizations can cause space leaks by remembering too much is old, but only recently appreciated [9, 11, 2]. The Chalmers Lazy-ML compiler [8] and the SML/NJ compiler [5] are the only ones we know of that guarantee "space safety."

Globalization: Local variables of different functions with nonoverlapping live ranges can be allocated to the same register or global without any save or restore [18, 12].

A stack of regions: Tofte and Talpin have demonstrated an analysis that can avoid garbage collection entirely [35], but unfortunately it does not satisfy the "safe for space complexity" rule.

8 Conclusions

Our new closure conversion algorithm is a great success. The closure conversion algorithm itself is faster than our previous algorithm. It makes every program smaller (by an average of 20%) and faster (by an average of 17% over many benchmarks). It decreases the rate of heap allocation by 36%, and (by obeying the "safe for space complexity" rule and keeping closures small) helps reduce the amount of live data preserved by garbage collection.

The closure analysis technique introduced in this paper can also be applied to compilers that do not use CPS as their intermediate languages. Both safely-linked closures and good use of callee-save registers are essential in building any real efficient compilers that want to satisfy the "safe for space complexity" rule.

Acknowledgement

We would like to thank Trevor Jim, Xavier Leroy, John Reppy, Jean-Pierre Talpin, and the LFP program committee for comments on an early version of this paper.

References

[1] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letter*, 25(4):275–79, 1987.

[2] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] Andrew W. Appel and Trevor Jim. Optimizing closure environment representations. Technical Report 168, Dept. of Computer Science, Princeton University, Princeton, NJ, 1988.

[4] Andrew W. Appel and Trevor Jim. Continuation-passing, closure-passing style. In *Sixteenth ACM Symp. on Principles of Programming Languages*, pages 293–302, New York, 1989. ACM Press.

[5] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In Martin Wirsing, editor, *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, pages 1–13, New York, August 1991. Springer-Verlag.

[6] Andrew W. Appel and Zhong Shao. Callee-save registers in continuation-passing style. *Lisp and Symbolic Computation*, 5(3):191–221, 1992.

[7] Andrew W. Appel and Zhong Shao. An empirical and analytic study of stack vs. heap cost for languages with closures. Technical Report CS-TR-450-94, Princeton University, Department of Computer Science, Princeton, NJ, March 1994.

[8] Lennart Augustsson. Garbage collection in the $<\nu, g>$-machine. Technical Report PMG memo 73, Dept. of Computer Sciences, Chalmers University of Technology, Goteborg, Sweden, December 1989.

[9] Henry G. Baker. The buried binding and stale binding problems of LISP 1.5. unpublished, undistributed paper, June 1976.

[10] Luca Cardelli. Compiling a functional language. In *Proc. of the 1984 ACM Conference on Lisp and Functional Programming*, pages 208–217, August 1984.

[11] David R. Chase. Safety considerations for storage allocation optimizations. In *Proc. ACM SIGPLAN '88 Conf. on Prog. Lang. Design and Implementation*, pages 1–9, New York, June 1988. ACM Press.

[12] Fred C. Chow. Minimizing register usage penalty at procedure calls. In *Proc. ACM SIGPLAN '88 Conf. on Prog. Lang. Design and Implementation*, pages 85–94, New York, June 1988. ACM Press.

[13] William D Clinger, Anne H Hartheimer, and Eric M Ost. Implementation strategies for continuations. In *1988 ACM Conference on Lisp and Fucntional Programming*, pages 124–131, New York, June 1988. ACM Press.

[14] G. Cousineau, P. L. Curien, and M. Mauny. The categorical abstract machine. In J. P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture, LNCS Vol 201*, pages 50–64, New York, 1985. Springer-Verlag.

[15] Olivier Danvy. Memory allocation and higher-order functions. In *Proceedings of the SIGPLAN'87 Symposium on Interpreters and Interpretive Techniques*, pages 241–252. ACM Press, June 1987.

[16] Amer Diwan, David Tarditi, and Eliot Moss. Memory subsystem performance of programs using copying garbage collection. In *Proc. 21st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 1–14. ACM Press, 1994.

[17] Lal George, Florent Guillaume, and John Reppy. A portable and optimizing backend for the SML/NJ compiler. In *Proceedings of the 1994 International Conference on Compiler Construction*, page (to appear), April 1994.

[18] Carsten K. Gomard and Peter Sestoft. Globalization and live variables. In *Proceedings of the 1991 Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, pages 166–177. ACM Press, June 1991.

[19] Jr. Guy L. Steele and Gerald Jay Sussman. The dream of a lifetime: A lazy variable extent mechanism. In *Proceedings of the 1980 LISP Conference*, pages 163–172, Stanford, 1980.

[20] Chris Hanson. Efficient stack allocation for tail-recursive languages. In *1990 ACM Conference on Lisp and Fucntional Programming*, pages 106–118, New York, June 1990. ACM Press.

[21] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. In *Proc. ACM SIGPLAN '90 Conf. on Prog. Lang. Design and Implementation*, pages 66–77, New York, 1990. ACM Press.

[22] Thomas Johnsson. Lambda Lifting: Transforming Programs to Recursive Equations. In *The Second International Conference on Functional Programming Languages and Computer Architecture*, pages 190–203, New York, September 1985. Springer-Verlag.

[23] Norman P. Jouppi. Cache write policies and performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 191–201. ACM Press, May 1993.

[24] Richard Kelsey and Paul Hudak. Realistic compilation by program transformation. In *Sixteenth ACM Symp. on Principles of Programming Languages*, pages 281–92, New York, 1989. ACM Press.

[25] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams. ORBIT: An optimizing compiler for Scheme. *SIGPLAN Notices (Proc. Sigplan '86 Symp. on Compiler Construction)*, 21(7):219–33, July 1986.

[26] David Kranz. *ORBIT: An optimizing compiler for Scheme*. PhD thesis, Yale University, New Haven, CT, 1987.

[27] P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–20, 1964.

[28] Xavier Leroy. Unboxed objects and polymorphic typing. In *Nineteenth Annual ACM Symp. on Principles of Prog. Languages*, New York, Jan 1992. ACM Press.

[29] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts, 1990.

[30] Guillermo Juan Rozas. Liar, an algol-like compiler for scheme. S.B. thesis, MIT Dept. of Computer Science and Electrical Engineering, June 1984.

[31] Barbara G. Ryder and Marvin C. Paull. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18(3):277–316, September 1986.

[32] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, Pittsburgh, PA, May 1991. CMU-CS-91-145.

[33] Guy L. Steele. Rabbit: a compiler for Scheme. Technical Report AI-TR-474, MIT, Cambridge, MA, 1978.

[34] Robert E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Science*, 9(3):355–365, December 1974.

[35] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value λ-calculus using a stack of regions. In *Proc. 21st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pages 188–201. ACM Press, 1994.

[36] David M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. MIT Press, Cambridge, MA, 1986.