Efficient and Safe-for-Space Closure Conversion

ZHONG SHAO
Yale University
and
ANDREW W. APPEL
Princeton University

Modern compilers often 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 the compilation of functional languages. This paper presents a new algorithm that exploits the use of compile-time control and data flow information to optimize function calls. 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 and global variables by 43%; and improves the already efficient code generated by an earlier version of 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.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*procedures, functions, and subroutines*; D.3.4 [**Programming Languages**]: Processors—*compilers; optimization; closure conversion*

General Terms: Languages, Algorithms, Performance, Theory

Additional Key Words and Phrases: closure conversion, closure representation, compiler optimization, space safety, callee-save registers, flow analysis, heap-based compilation

1. INTRODUCTION

Compilers for higher-order languages (e.g., Scheme, ML, Haskell) take great efforts to optimize function calls and returns because they are fundamental control structures. Before a function call occurs, 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 [Steele 1978].

In a CPS-based compiler, a closure environment is constructed at each function-definition site; it provides runtime access to bindings of free variables in the function 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

Authors' addresses: Zhong Shao, Department of Computer Science, Yale University, 51 Prospect Street, New Haven, CT 06520-8285; Email: shao-zhong@cs.yale.edu; Andrew W. Appel, Department of Computer Science, Princeton University, 35 Olden Street, Princeton, NJ 08544-2087; Email: appel@cs.princeton.edu. Preliminary presentations of this work appeared in Proceedings of the 1994 ACM Conference on Lisp and Functional Programming [Shao and Appel 1994] and in the first-named author's Ph.D. dissertation [Shao 1994]. This research was supported by the National Science Foundation Grants CCR-9002786, CCR-9200790, and CCR-9501624.

implemented in the same way because they are essentially calls to continuation functions if represented in CPS. A non-CPS-based compiler does not introduce explicit continuations, but it requires a similar mechanism to manage all the closures.

The goal of closure conversion is to choose closure representations that minimize stores (for closure creation), fetches (to access free variables), and memory use (for reachable data). Depending on the runtime behavior of each function, closures can be represented as data structures of virtually any shape—allocated in the heap, on the stack (if any), or in registers. Clearly, the decisions of where and how to represent closures at runtime can greatly affect the quality of generated code. For example, Kranz's Orbit [Kranz et al. 1986; Kranz 1987] compiler generates very fast code by having six different closure allocation strategies for different kinds of functions; the MIT scheme compiler [Rozas 1984; Hanson 1990] implements tail-recursive procedure calls as efficiently as most explicit iteration constructs by using special closure representations and calling conventions; the Standard ML of New Jersey compiler (SML/NJ) [Appel and MacQueen 1991; Appel and Shao 1992] represents the return-continuation closure using a set of callee-save registers to achieve closure sharing and fast access to free variable bindings. In our current work, we have integrated all of these techniques into a unified framework to achieve excellent code quality.

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 are also allocated in the heap, they can be shared with other heap-allocated closures. This is impossible under stack allocation because stack frames often have shorter lifetime than heap-allocated closures.

— Tail recursive calls—which are often quite troublesome to implement correctly on a stack [Hanson 1990]—can be implemented very easily.

— All of our closure analysis and optimizations can be cleanly represented using continuation-passing and closure-passing style [Appel and Jim 1989] 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.

— Because all closures are either allocated in the heap or in registers, first class continuations (i.e., *call/cc*) are very easy to support efficiently.

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 locality of reference, 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, while true, are not as significant as one might think, and are offset by the disadvantages of stack allocation:

—As we will show in Section 4, because most parts of continuation closures are allocated in callee-save registers [Appel and Shao 1992], 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 [Appel and Shao 1996].

—In a companion paper [Appel and Shao 1996], 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 [Jouppi 1993; Diwan et al. 1994].

—The amortized cost of collection can be very low [Appel 1987; Appel and Shao 1996], especially with modern generational garbage collection techniques [Ungar 1986; Reppy 1993].

The main contribution of this paper is an *efficient* and *safe-for-space* closure conversion algorithm that integrates and improves most previous closure-analysis techniques [Kranz 1987; Appel and Shao 1992; Steele 1978; Rozas 1984; Hanson 1990; Johnsson 1985] using a simple and general framework expressed in continuation-passing and closure-passing style [Appel and Jim 1989; Appel and Shao 1992]. 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 an earlier version of the SML/NJ compiler by about 17% on a DECstation 5000.

2. EFFICIENCY AND SPACE SAFETY

It is difficult to design good closure-allocation schemes that are both time-efficient and space-efficient. In fact, optimization of closure representations is sometimes dangerous and even unsafe for *space usage* (i.e., the maximum size of simultaneously live data during execution). In 1988, Chase [1988] observed that certain storage-allocation optimizations may convert a program that runs robustly into one that does not, due to the requirement of a larger fraction of memory than the program actually needs. Appel [1992] also noticed that programs using linked closures[1], or stack-allocated activation records, may cause a compiled program to use much more memory.

[1] A *linked closure* [Landin 1964] is a record that contains the bound variables of the enclosing function, together with a pointer to the enclosing function's closure.

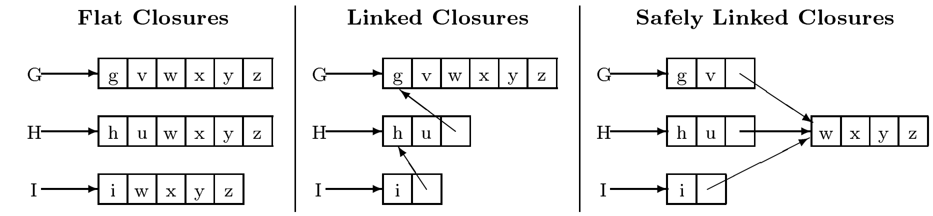

Fig. 1. A comparison of three closure representations

```
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 nil 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,nil)
```

Fig. 2. An example in Standard ML

Consider the program in Figure 2 written in Standard ML (SML) [Milner et al. 1990]. With flat closures[2] (see Figure 1), each evaluation of `f(...)()` yields a closure H for function `h` that contains just a few integers `u`, `w`, `x`, `y`, and `z`; the final result (i.e., `result`) contains N copies of the closure for `h`, thus it uses at most $O(N)$ space. With linked closures (also see Figure 1), each closure 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 uses $O(N^2)$ space instead of $O(N)$. Apparently, this space leak is caused by inappropriately retaining some "dead" objects (i.e., `v`) that should be reclaimed earlier by the garbage collector.

In 1992, we found several instances of real programs whose live data size (and

[2] A *flat closure* [Cardelli 1984] is a record that holds only the free variables needed by the function.

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 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.* Please see Appel [1992] for a more formal definition.

The SSC definition of reachability is often viewed as an optimization to eliminate environment-related space leaks, but we believe that it is a crucial property that any industrial-strength compiler for functional languages must satisfy. Consider the following ML function simulating N phases of a typical compiler:

```
fun compile (codeInSrc) =
  let val codeInIL1 = phase1(codeInSrc)
      val codeInIL2 = phase2(codeInIL1)
      ......
      val codeInILN = phaseN(codeInILNm1)
   in codeInILN
  end
```

If we rewrite this code in C, we would manually use the *free* operations to deallocate all intermediate data structures (e.g., after a compilation phase is done). In a garbage-collected language, however, we no longer have the capability to manually free memory, since memory management is now done implicitly by the garbage collector. Naturally, we would hope that the compiler would be responsible for getting everything right.

Unfortunately, most compilers treat the `compile` function just like a simple C-like subroutine. Local variables such as `codeInSrc`, `codeInIL1`, ..., and `codeInILN`—assuming they are all pointers to some heap-allocated objects—are all placed in the same stack frame, and they are considered to be *live* until after we return from the call to `phaseN` and when we pop off the stack frame for `compile`. This is clearly unacceptable because it keeps all intermediate representations simultaneously live, thus requires much more memory than really necessary. A C programmer would have manually freed the intermediate representations from the earlier phases before moving onto the late phases.

We thus argue that garbage-collected languages must satisfy the new SSC scoping rule if they use functions pervasively. Each local variable should be considered "dead" after its *last* use in the current function body. By dead, we really mean that it is *not contributing to the liveness of the data structure that it points to.* Supporting the SSC rule is important because functions such as `compile` are common in large software. The SSC rule is not as important for C-like languages because we can manually deallocate intermediate data structures in the program source.

Traditional stack-allocation schemes and linked closures obviously violate the SSC rule because local variable bindings will stay on the stack until they exit their scope, so they 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 [1988] also violate the SSC rule.

Obeying the SSC rule 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 clear whether there are other references to the old closure. The challenge is to find efficient closure strategies that obey the SSC rule while minimizing copying.

Our new closure-conversion algorithm uses *safely linked closures* (the 3rd column in Figure 1) that contain only variables actually needed in the function but avoid closure copying by grouping variables with the 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 (one layer for the closure itself; another for records of different life time), so they still enjoy very fast variable access time.

To support the SSC scoping rule for cases like the `compile` function, we treat all activation records as closure environments for continuation functions. As for flat closures, we insist that each such environment contain only variables used in the continuation. To avoid high copying overhead, we use a combination of safely linked closures and callee-save registers to represent such environment; the resulting code not only satisfies the SSC rule (thus minimize the memory use) but also supports very fast closure creation and free-variable access.

An alternative way is to stick to the standard stack-based scheme, which violates the SSC rule because dead local variables remain in the stack frame until the function returns. This can be partially fixed by associating a descriptor with each return address, showing which variables are live in the continuation [Appel and Shao 1996; Augustsson 1989]; the garbage collector then finds and interprets the descriptor during each collection to ignore tracing those "dead" local variables. In addition, free variables inside a function closure must be copied from the heap to the stack in order to fully obey the SSC rule. For example, in the following ML program:

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

The closure for function `g` contains free variables `h`, `u`, `v`, and `w`. Under the standard stack implementation, `g`'s frame would contain a pointer to `h`'s closure, and all four free variables of `g` are kept live until `g` returns. But variable `u` reaches its last use at the first call to `h`; keeping it live with the rest of the closure is clearly not space-safe.

```
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)
  end
```

Fig. 3. Function iter in Standard ML

In a real compiler, the stack-based scheme must also incorporate a wide range of other extensions to support tail recursion [Hanson 1990], generational stack collection [Cheng et al. 1998], efficient call/cc [Clinger et al. 1988; Hieb et al. 1990], and exception handling [Ramsey and Peyton Jones 1999]. Each of these may require non-trivial changes to the basic scheme such as sharing activation records, inserting special markers in the middle of a frame, and randomly updating the return address. The main challenge for the stack-based implementation is thus on how to maintain and reason about the SSC property in the presence of these extensions. We believe this is a very difficult problem; in fact, we are not aware of any compiler that supports all of these extensions, not to mention enforcing the SSC rule.

Because of this difficulty with stacks and also for reasons given in the companion paper [Appel and Shao 1996], we do not use any runtime stack. Instead, we treat all activation records as closures for continuation functions and allocate them in registers or in the heap. Without the stack, supporting tail recursion, generational garbage collection, efficient call/cc, and exception handling all becomes straightforward [Appel 1992]. It is also easy to maintain the SSC property because all control transfers are now implemented as plain function calls; as long as each closure contains only free variables that are live in the target function body, and garbage collection is only invoked at the beginning of every function call, a local variable binding will be unreachable after its last use within its scope.

The rest of this paper is organized as follows. We first use examples to review the continuation-passing and closure-passing intermediate languages and to show how activation records might be allocated in the heap and in callee-save registers (Section 3). We then present our detailed closure-conversion algorithm (Section 4). We apply the algorithm to several common program fragments and show how it achieves the desirable results (Section 5). We also give our measurement data and show how well our algorithm works on realistic benchmarks (Section 6). Finally, we discuss the related work and then conclude.

3. CONTINUATIONS AND CLOSURES

In this section, we illustrate the CPS-based compilation [Steele 1978; Kranz 1987; Appel 1992]) and our new closure-analysis algorithm on the example in Figure 3. The function iter iteratively applies function f to argument x until it converges to satisfy predicate p.

We use CPS as the intermediate language for closure conversion because we want to treat all activation records as continuation closures. An alternative is to use the A-normal form [Flanagan et al. 1993] extended with the return-address labels; this is essentially just a syntactic variant of CPS. We are not advocating using CPS as the

v	::=	*variable*	a	::=	$v \mid i \mid b$
i	::=	*integer constant*	p	::=	*primitive operator*
b	::=	*boolean constant*			
			d	::=	`fun` $v(v_1,v_2,\ldots,v_n) = e$
e	::=	`let` d `in` e `end`		$\mid$	`val` v = `select`(i,a)
	$\mid$	`if` a `then` e_1 `else` e_2		$\mid$	`val` $v = (a_1,a_2,\ldots,a_n)$
	$\mid$	$a(a_1,a_2,\ldots,a_n)$		$\mid$	`val` $v = p(a_1,a_2,\ldots,a_n)$

Fig. 4. Abstract syntax of CPS

```
fun iter(C,x,p,f) =
  let fun h(K,a,r) =
        let fun J(z) = if z then K(a)
                       else let fun Q(b) = h(K,b,a)
                             in f(Q,a)
                            end
         in p(J,a,r)
        end
   in h(C,x,1)
  end
```

Fig. 5. Function `iter` after CPS conversion

intermediate language for standard data-flow and control-flow optimizations [Appel 1992]. In fact, recent versions of the SML/NJ and FLINT compilers [Shao 1997] chose to use the A-normal form as the intermediate language for optimizations and then convert it into CPS for closure conversion.

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 4. 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 record construction, field selection, primitive operation, and function definition must be followed by a continuation expression e (through the `let` expression). On the other hand, function application $a(a_1, \ldots, a_n)$ does not specify a continuation expression because in CPS, functions never *return* in the conventional sense. Instead, it is expected that each user function will take a return-continuation as one of its arguments. The continuation is defined in the ordinary way (using `fun`), and will presumably be invoked by the callee in order to continue the computation.

To simplify the syntax further, later in the paper, we use `let` d_1 d_2 ... d_n `in` ... `end` to denote a sequence of `let` expressions, i.e., `let` d_1 `in let` d_2 `in` ... `let`

```
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)
  end
```

Fig. 6. Function `iter` after CPS-based optimizations

d_n `in` ... `end` ... `end end`. To write less verbose code, we often combine primitive operations with other expressions (e.g., `3+4+5` or `if x=0 then ...`).

Figure 5 shows the code of the function `iter` after translation into CPS. Figure 6 shows the code `iter` after the continuation argument `K` of `h` has been hoisted out of the loop because it is loop-invariant (it is always `C`). Such optimizations are performed after CPS-conversion, but before the closure analysis which 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 as *user functions* (e.g., `iter`, `h`), and those introduced by CPS conversion as *continuation functions* (e.g., `J`, `Q`). *Continuation variables* are all those formal parameters (normally placed as the first argument of a user function) 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 of their call sites. 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 [1987] argued that different kinds of functions should use different closure-allocation strategies. For example, the closure for a known function (e.g., `h` in Figure 6) 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. The closure for an escaping function, on the other hand, may have to be allocated as a heap record which contains both the machine code address of the function plus bindings to all its free variables.

```
fun iter(I,C0,C1,C2,C3,x,p,f) =
  let fun h(a,r,CR,p) =
        let fun J0(J1,J2,J3,z) =
              if z then
                let val C0 = select(0,J1)
                    val C1 = select(1,J1)
                    val C2 = select(2,J1)
                    val C3 = select(3,J1)
                 in C0(C1,C2,C3,J2)
                end
              else
                let fun Q0(Q1,Q2,Q3,b)
                              = h(b,Q2,Q1,Q3)
                    val f = select(4,CR)
                    val f0 = select(0,f)
                 in f0(f,Q0,J1,J2,J3,J2)
                end
            val p0 = select(0,p)
         in p0(p,J0,CR,a,p,a,r)
        end
      val CR = (C0,C1,C2,C3,f)
   in h(x,1,CR,p)
  end
```

Fig. 7. Function `iter` after closure conversion

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 CPS intermediate representation. We did so as follows [Appel and Shao 1992]: each CPS-converted user function f is passed its ordinary arguments, a continuation function c_0, and k extra arguments $c_1, ..., c_k$. The function "returns" by invoking c_0 with a "result" argument r and the additional arguments $c_1, ..., c_k$. Thus, the "callee-save" arguments $c_1, ..., c_k$ are handed back to the continuation. When this CPS code is translated into machine instructions, $c_1, ..., c_k$ will stay in registers throughout the execution of f, unless f requires 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 [Appel and Shao 1992], we outlined this framework and demonstrated that it could reduce allocation and memory traffic. However, we did not have an algorithm to fully exploit the flexibility that callee-save registers provide.

Closure creation and use can also be represented using the CPS language itself [Appel and Jim 1989; Kelsey and Hudak 1989]. 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 7 lists the code of function `iter` after translation into CLO. Continuation functions and variables (e.g., `C,J,Q`) are 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 6) had free variables `C,f,a,h.` With three callee-save registers, `C` is replaced by four new 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 the four parameters `J0-J3`. Clearly, if there are more than three free variables, some of the callee-save arguments must be boxed and allocated in the heap. For our example, we build closure `CR` which is bound to `J1` in the call on line 19.

Previous closure conversion algorithms [Steele 1978; Kranz et al. 1986; Appel and Jim 1989] 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

This section presents our new closure-conversion algorithm using the framework defined in Section 3. Our algorithm takes a closed CPS expression e as the input, decides the closure representation for each function definition (in e), and then transforms e into closure-passing style (CLO).

As we argued at the beginning of Section 3, we use CPS as the intermediate language because we want to treat activation records as continuation closures. Using CPS makes it easier to enforce the SSC property. For example, to have a closure-conversion algorithm satisfy SSC, all we need is to make sure that the closure environment for each function—whether it is a user function or a continuation—-only contains variables that are truly free in the function body. This is because all control transfers in CPS are realized as function calls; at any program point, a local variable that is no longer live can not be free in the current continuation, and vice versa. If the compiler can only invoke garbage collection at the beginning of a function (which is true for SML/NJ [Appel 1992]), there is no need to impose any additional requirement inside the function body.

The simplest way to enforce SSC is to always use flat closures and to allocate everything on the heap. This is clearly inefficient because building flat closures involves excessive copying, and memory access is much slower than reading from registers. Our main idea is to have each closure contain the same set of free variables (as under the flat representation) but use a new layout that facilitates sharing and takes advantage of registers. Our key technique is to exploit the compile-time data-flow and control-flow information to build safely linked closures (see Section 2) and to allocate continuation closures in callee-save registers (see Section 3.2). Our

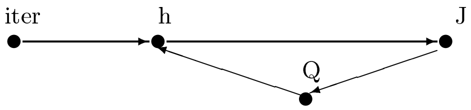

Fig. 8. The extended CPS call graph for function `iter`

algorithm satisfies the SSC rule because regardless of what optimizations we do, we insist that the closure for each function only contain its own free variables.

Our algorithm does not use any runtime stack but our main technique (i.e., using data-flow and control-flow information to build safely linked closures and to allocate closures in registers) is also applicable to stack-based implementations. In fact, one can still use CPS as the intermediate language while supporting stack allocation [Kranz 1987; Morrisett et al. 1998].

In the rest of this section, we present our new closure-conversion algorithm in detail. Our algorithm consists of four steps, each of which is implemented as a separate pass over the abstract syntax or the call graph:

(1) calculate the control-flow information by constructing the call graph;
(2) find the set of raw free variables and their lifetime information;
(3) run closure-strategy analysis to determine where to allocate each closure;
(4) decide the closure layout and transform the program into closure-passing style.

Among these different procedures, the only thing that affects the SSC property is whether the closure for each function only contains its own free variables. Everything else we do, such as calculating the call graph, finding the variable lifetime information, and the closure-strategy analysis, are heuristics for identifying more closure sharing and for making better use of registers.

4.1 Extended CPS call graph

A closure for a function must contain all of its free variables, however, different variables might be used along different program paths (i.e., conditional branches). How frequently these paths are executed at runtime can affect the optimal closure layout. For example, the continuation function `J` in Figure 6 contains two branches:

```
fun J(z) = if z then C(a)
           else f(Q, a)
```

Clearly, `C` is free in the `then` branch and `f` is free in the `else` branch. If we only have one register to spare, we would accommodate `f` first because the `else` branch happens to be in the middle of a loop.

Our first step is thus to calculate the control-flow information of the input program. We are interested in using such information to identify loops and to do static branch prediction [Ball and Larus 1993].

Every function definition in CPS ends with a function call. We say f *directly calls* g if in the body of f, there exists a path that ends with a call to g. For example, in Figure 6, continuation `J` directly calls `C` and `f` but not `h` even though the call to `f` would invoke `Q` and then indirectly `h` as well.

Given a CPS expression e, its extended CPS call graph is a directed graph with the set of its function-definition variables as nodes and with edges from v_1 to v_2 if (1) v_1 directly calls v_2, or (2) v_1 directly calls some function with v_2 as its return continuation. For each node v, we also define v's predecessor set, $\mathit{pred}(v)$, as the set of all nodes that have an edge pointing to v. Figure 8 gives the extended CPS call graph for the function `iter`. Although `J` is not directly called by `h`, we conservatively assume that `p` will eventually call its return continuation `J`. The predecessor set of `h` contains two elements, `iter` and `Q`.

The extended CPS call graph captures a simple set of control-flow information.[3] Cycles in the graph imply loops or recursions; for example, for the graph in Figure 8, the path from `h` to `J` to `Q` and back to `h` forms a loop. The nesting hierarchy of loops can be revealed by running the interval analysis [Muchnick 1997; Ryder and Paull 1986; Tarjan 1974]. Intuitively, an interval is either a natural loop, a maximal acyclic subgraph, or a minimal irreducible region; the interval analysis partitions the set of nodes into disjoint intervals, with each interval representing a proper loop layer or an irreducible region.

Intuitively, an interval is either a natural loop, a maximal acyclic subgraph, or a minimal irreducible region; the interval analysis partitions the set of nodes into disjoint intervals, with each interval representing a proper loop layer.

Given a function definition f in a CPS program e, we define f's *loop level*, $L(f)$, as the nesting depth of its interval in the extended CPS call graph of e, assuming the outmost interval is at depth 0. For any variable that does not directly define a function (e.g., `C,f,p` in Figure 6), its loop level is assigned to be 0. The loop level of each call from f to g is defined as:

$$L(f,g) = \mathit{min}(L(f), L(g)).$$

Given a CPS expression defined inside the body of function f, we deduce its loop level based on the following inductive procedure:

—$L(\texttt{if}\ a\ \texttt{then}\ e_1\ \texttt{else}\ e_2) = \mathit{max}(L(e_1), L(e_2))$;

—$L(\texttt{let}\ d\ \texttt{in}\ e\ \texttt{end}) = L(e)$;

—$L(a(a_1,\ldots,a_n))$=$L(f,a)$ if a is a continuation; and $\mathit{max}(L(f,\ a),\ L(f,\ a_1))$ if a is a user function and a_1 is its return continuation.

The loop-level number is used to assign a priority to each program path—a higher number means it is nested deeper inside loops thus it could be taken more often at runtime. For example, in function `iter`, the loop level for variables `iter`, `C`, `f`, and `p` is 0; the loop level for `h`, `J`, and `Q` is 1. In the conditional statement `if z then ... else ...` inside the definition of `J`, the `then` branch calls continuation `C` and the `else` branch calls `f` and `Q`. Because $L(\texttt{J},\texttt{Q}) = \mathit{min}(L(J), L(Q)) = 1$ and $L(\texttt{J},\texttt{C}) = 0$, so the `else` branch is more likely to be taken. The final closure layout should put a higher priority on making the `else` branch run faster.

4.2 Raw free variables with lifetime

To implement the safely-linked closures, we want to group variables into closure records if they have similar lifetime. If variable v_1 is defined much later than v_2,

[3] Shivers [1991] presented several techniques that can find more accurate control-flow information.

Table I. Raw free variables and closure strategies

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

v_3, and v_4, then we may not have enough registers to hold v_2 to v_4 while waiting for v_1. If we save v_2 to v_4 into a closure, and v_2's last use is much earlier than v_3's or v_4's, then sharing this closure might not obey the SSC rule.

Most closure-conversion algorithms [Appel 1992; Kranz 1987; Steele 1978] start by calculating the set of free variables for each function definition. These free variables are called *raw free variables* because they may be substituted by a set of other variables during the translation. For example, if g is free inside f, and g corresponds to a known-function definition, then the closure for f must also contain g's free variables. We use *true free variables* to denote the set of variables that will be finally put into the actual closure.

The second step of our algorithm is thus to calculate the set of raw free variables together with their lifetime information. For the purpose of closure conversion, variables that are live in the same set of functions can be treated as having the same lifetime. We use *stage number* to measure such function-level lifetime. Given a function definition f, its stage number, $SN(f)$ is defined as follows:

—$SN(f) = 1$ if f is the outmost function;

—$SN(f) = 1 + SN(g)$ if f is a user function and g immediately encloses f;

—$SN(f) = 1 + max\{SN(g) \mid g \in pred(f)\}$ if f defines a continuation.

Given a CPS call graph G, because continuation functions are never recursive or mutually recursive, the continuation subgraph of G (i.e., by including only the continuation nodes) must be free of cycles. The stage number thus captures the order of all function-return points in the original program. In addition, it also captures the "enclosing" (or nesting) relationship for all user functions.

We can now define the *use time* for each use of a variable v as $SN(f)$ where f is the nearest enclosing function definition (for this use of v). Given a function definition f, its raw free-variable information is 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 for 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 take advantage of the control-flow information (see Section 4.1), we can also adjust the lut and fut numbers by assigning priorities to different uses along different program paths. For example, for a CPS expression **`if`** a **`then`** e_1 **`else`** e_2, we can ignore all uses of v in e_1 if $L(e_1) > L(e_2)$. The bias towards the loop edges can lead to more efficient closure representations at runtime.

Table I gives the stage number and the set of raw free variables for all functions defined in Figure 6. Here, `iter` is numbered as stage 1 because it is the outmost function; `h` is nested immediately inside so it is numbered as 2; the predecessor set of `J` contains `h` only so `J` is numbered as 3; and similarly, `Q` is numbered as 4. For raw free variables, `Q` uses two free variables `a` and `h` both at stage 4; `J` uses `C`, `f`, and `a`, thus the first-use number for `a` is 3 while the last-use number is 4. Clearly the same variable can have different *lut* and *fut* numbers inside different function definitions (e.g., `a` in `J` vs. `a` in `Q`).

4.3 Closure-strategy analysis

The third step of our algorithm, also called "closure-strategy analysis," is to determine where in the machine to allocate each closure. Because we do not use any runtime stack, all we need is to decide how many *slots* (i.e., registers) are allocated to each closure; this number is denoted as $S(f)$ for each function definition f. We calculate $S(f)$ using the algorithm described below.

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 [Appel and Shao 1992]. In special cases, some continuation functions can be represented differently; we will discuss this more in Section 5.3.

If f is a known function, then its call sites are all explicitly known at compile time. The closure environment for f can be all allocated in registers. This is not always desirable because (1) there are only a limited number of registers on the target machine, and (2) allocating all free variables in registers can add pressures to functions that have f as a free variable. Either of these can create more register-memory traffic (see Section 5.2). We use the following algorithm to resolve these conflicts:

—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 in function f (assuming they will be passed in registers);

—Then, for each known function f, we first find all those functions that not only call f but also have f as one of their free variables; this set is denoted as V:

$$V = \{g \mid g \in \mathit{pred}(f) \text{ and } f \text{ is free in } g\}$$

If function g calls function f and f is free in g, then the number of slots assigned to f should not be larger than the number of slots available inside g, otherwise it will lead to spilling. Suppose j is the number of variables that are free in g but not in f; then it is a good idea to assign f with $S(g) - j$ slots; we use $T(g)$ to denote this constraint:

$$T(g) = \mathit{max}(1, S(g) - j)$$

We then refine the number of slots assigned to f as:

$$S(f) = \mathit{min}(\{T(g) \mid g \in V\} \cup \{S(f)\})$$

This procedure is repeated until the slot number assigned to every known function no longer changes.

This algorithm always terminates and reaches a fix point because both $T(g)$ and $S(f)$ are always positive numbers, and the sum of $S(f)$ for all known functions gets smaller in each iteration. If K is the number of registers on the target machine, and N is the number of function definitions, the algorithm could take $O(KN)$ rounds to finish in the worst case. In practice, it takes no time and reaches the fix point in less than three rounds.

We can again take advantage of the control-flow information when calculating $S(f)$. For all functions g in the above set V, we can give a higher priority to those functions with higher $L(g, f)$ values because a call to these functions happens to fall inside deeper loops.

Let's apply this algorithm to our example in Figure 6. Function `iter` is an escaping user function, so it is assigned one slot. Suppose we use three callee-save registers, all escaping continuations are assigned three slots, that is, both $S(\texttt{Q})$ and $S(\texttt{J})$ should be equal to 3. Function `h` is initially assigned with 14 slots, assuming we have 16 available registers on the target machines. The predecessor set for `h` contains two functions, `iter` and `Q`, but `h` is not free in `iter`, so the set V for `h` only contains `Q`. Because $S(\texttt{Q}) = 3$, and only one variable, `a`, is free in `Q` but not in `h`, so $T(\texttt{Q}) = max(1, 3 - 1) = 2$. Function `h` is thus assigned with $S(\texttt{h}) = min(T(\texttt{Q}), 14) = 2$ slots (see Table I).

It is important to note that by assigning two slots to `h`, we successfully avoid building any closure inside the loop (see Figure 7, lines 02-20). Naive lambda-lifting and closure-analysis algorithms [Johnsson 1985; Appel 1992] would have allocated all of `h`'s free variables into registers; this would force the continuation `Q` to spill and allocate a closure in the heap every iteration.

4.4 Translation into closure-passing style

The last step of our algorithm is to translate the input CPS program into closure-passing style. During the translation, we run the closure-representation analysis to decide the final layout for each closure. More specifically, given a function f, if f contains m free variables and is assigned n slots, how do we place these m values into n registers?

Given a CPS expression e, the translation into closure-passing style is done by processing each function definition through a preorder traversal of e. During the traversal, we maintain and update the following three environment structures:

whatMap. A mapping from functions processed so far to their closure layout.

whereMap. A list of currently visible closures and values;

baseRegs. The current contents of callee-save registers.

We processing each function definition f, we execute the following tasks:

4.4.1 *Finding the transitive closure of raw free variables.* We check if f is recursive or mutually recursive with other functions. Only user functions can be recursive or mutually recursive; continuation functions are never recursive. The language defined in Figure 4 does not support mutual recursion directly, but in real intermediate languages [Appel 1992], they are introduced via a single `fun` declaration. In this case, a function declaration simultaneously defines n functions f, g, h,

...; they are processed together and eventually share one closure.[4] If f calls g and g calls h, then the free variables of f must also contain those for g and h, and so on. For this reason, we need to calculate the transitive closure of the raw free-variable information. We use the following algorithm:

—Suppose the set of raw free variables of f—calculated by the algorithm in Section 4.2—is named as $RFV(f)$; also suppose f recursively calls g, then $RFV(f)$ must contain a triple (g, fut_g, lut_g). We calculate the transitive closure by merging $RFV(f)$ with the following set:

$$\{(v, fut_g, lut_g) \mid v \in RFV(g)\}$$

When merging all triples for the same variable, we take the minimum of all the *fut* numbers and the maximum of all the *lut* numbers.

—We repeat this procedure until it reaches a fix point. This could take $O(n^2)$ rounds in the worst case, but it usually takes two to three rounds in practice.

We use $RFV^*(f)$ to denote the final transitive closure for function f. Let's apply this algorithm to our example in Figure 6. Here, function `h` is recursive and as shown in Table I, $RFV(\mathtt{h})$ is initially equal to:

$$RFV(\mathtt{h}) = \{(\mathtt{p}, 2, 2), (\mathtt{C}, 3, 3), (\mathtt{f}, 3, 3), (\mathtt{h}, 4, 4)\}$$

Because `h` calls itself, we merge this with the set:

$$\{(\mathtt{p}, 4, 4), (\mathtt{C}, 4, 4), (\mathtt{f}, 4, 4), (\mathtt{h}, 4, 4)\}.$$

The final transitive closure $RFV^*(\mathtt{h})$ is equal to:

$$RFV^*(\mathtt{h}) = \{(\mathtt{p}, 2, 4), (\mathtt{C}, 3, 4), (\mathtt{f}, 3, 4), (\mathtt{h}, 4, 4)\}.$$

Because continuations are never recursive, the transitive closure for `J` and `Q` remains same as shown in Table I:

$$\begin{aligned} RFV^*(\mathtt{J}) &= RFV(\mathtt{J}) = \{(\mathtt{C}, 3, 3), (\mathtt{f}, 3, 3), (\mathtt{a}, 3, 4), (\mathtt{h}, 4, 4)\}; \\ RFV^*(\mathtt{Q}) &= RFV(\mathtt{Q}) = \{(\mathtt{a}, 4, 4), (\mathtt{h}, 4, 4)\}. \end{aligned}$$

4.4.2 *Calculating the true free variables.* Next we find the set of *true free variables*, $TFV(f)$, by replacing each continuation variable in $RFV^*(f)$ by its corresponding callee-save variables, and each function-definition variable (whether it defines a user function or a continuation function) by its closure contents (i.e., slot variables). For example, in $RFV^*(\mathtt{h})$ there is one continuation variable `C` and one function-definition variable `h`. We remove `h` because it is available inside its own body. Suppose we use three callee-save registers; we replace `C` by a code pointer `C0` and its three callee-save variables `C1`, `C2`, and `C3`. The set of true free variables $TFV(\mathtt{h})$ is equal to:

$$TFV(\mathtt{h}) = \{(\mathtt{p}, 2, 4), (\mathtt{C0}, 3, 4), (\mathtt{C1}, 3, 4), (\mathtt{C2}, 3, 4), (\mathtt{C3}, 3, 4), (\mathtt{f}, 3, 4)\}.$$

[4] The actual implementation would also divide these n functions into escaping and known functions. All escaping functions share one closure. All known functions can still allocate their environments in the registers.

Notice here all callee-save variables naturally inherit C's *fut* and *lut* numbers.

By the time we reach the continuations J and Q, we have already decided the layout for function h (see the rest of the algorithm below). For the translation in Figure 7, function h is assigned two slots: one slot contains the closure CR, another the variable p. So we replace each occurrence of h in $RFV^*(\mathtt{J})$ and $RFV^*(\mathtt{Q})$ by its slot variables. The true free variables $TFV(\mathtt{J})$ and $TFV(\mathtt{Q})$ are equal to:

$$\begin{aligned} TFV(\mathtt{J}) &= \{(\mathtt{C0},3,3),(\mathtt{C1},3,3),(\mathtt{C2},3,3),(\mathtt{C3},3,3),\\ &\quad\ (\mathtt{f},3,3),(\mathtt{a},3,4),(\mathtt{CR},4,4),(\mathtt{p},4,4)\};\\ TFV(\mathtt{Q}) &= \{(\mathtt{a},4,4),(\mathtt{CR},4,4),(\mathtt{p},4,4)\}. \end{aligned}$$

Here we didn't include the code pointer h because it is always available to functions in the same code segment. Again, C0-C3 inherit the lifetime from C, and the slot variables CR and p inherit from their corresponding function h.

4.4.3 *Sharing with closures in the current environment.* Given a function f, suppose $TFV(f)$ contains m variables, and f is assigned n slots by the closure-strategy analysis in Section 4.3. If $m \le n$, then we are done. Otherwise, we search through the current list of visible closures maintained by the **whereMap** environment, and see if there is any closure record that we can reuse. The sharing is *safe for space* as long as we only reuse those closures whose contents are a subset of $TFV(f)$; in order words, we only reuse a closure if it does not bring in additional new variables. Because we use safely linked closures, certain closure sharings have already been anticipated while processing the enclosing function definitions. If there are multiple sharable closures, we use the "best fit" heuristic to decide which one to reuse.

Let's look at our example of iter again. When we process function h, it has six free variables and is assigned two slots, but there are no closures accessible in the current environment, so no sharing is possible. For continuation J, however, we have already constructed a closure CR for the enclosing function h. As shown in Figure 7, the closure CR contains variables C0-C3 and f, all of which are elements of $TFV(\mathtt{J})$. Because J has eight free variables and is assigned three slots only, we reuse the closure CR, and reset $TFV(\mathtt{J})$ to be:

$$TFV(\mathtt{J}) = \{(\mathtt{a},3,4),(\mathtt{CR},3,4),(\mathtt{p},4,4)\}.$$

Notice that we set the *fut* number of CR to be 3, because that is when variables C0-C3 and f are first used. We need to do nothing on continuation Q because it has three variables and is assigned three slots. In the end, the closure CR is shared by all three functions, h, J, and Q.

4.4.4 *Allocating closures into registers.* After closure sharing, if the size m of $TFV(f)$ is still larger than n, we have to allocate parts of the closure in the heap. We do this by putting $n-1$ variables into one slot each and packing the remaining $m-n+1$ variables into the heap closure. We choose these $n-1$ variables based on the following criteria. First, we favor those variables with the smaller *lut* number because putting them in registers can avoid constructing extra closures. Second, we select those variables with the smaller *fut* number because this makes them easier to access. Third, we check if 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 unnecessary register moves.

In our example, function `h` is assigned two slots but `h` has six true free variables, so only one of them can be allocated in the register. We put variable `p` in the register because it has the smallest *fut* number (all variables have the same *lut* number).

4.4.5 *Creating safely linked closures.* Finally, we decide the layout for the spilled heap record—which contains those $m - n + 1$ variables—based on the *lut* numbers. The *lut* number indicates when a variable must be considered as dead. To allow other functions to share this closure while not violating the SSC rule, variables with distinct *lut* numbers should be put into a separate record.

For function `h` in our example, all five free variables `C0-C3` and `f` have the the same *lut* numbers. So we construct a single record `CR` as shown on Line 21 in Figure 7. The final environment layout for `h` is the closure `CR` plus the variable `p`. We finish processing `h` by updating the **whatMap**, **whereMap** and **baseRegs** environments accordingly based on this final representation.

In the example in Figure 1, the closure for `g` must split its free variables into two records because `v`'s *lut* number was different from those of `w,x,y,z`.

It is important to note that the *lut* numbers are not essential for the SSC rule even though we use them to decide the closure layout. Our algorithm satisfies the SSC rule because we never reuse a closure that brings in additional free variables (see Section 4.4.3). For the example in Figure 1, if we accidently assigned `v` the same *lut* number as `w,x,y,z`, the closure for `g` would put all five variables into a single record. This will prevent `h` from reusing `g`'s closure (because `v` is not free in `h`), but it won't destroy the SSC property.

4.4.6 *Finding the access path for non-local variables.* When translating the body of each function into closure-passing style, we need to replace each occurrence of a free variable v by its access path. This can be calculated by doing a breadth-first search of v in the **whereMap** environment. We use the "lazy display" technique used by Kranz [1987], so that loads of common paths can be shared.

Let's take the function `i` in Figure 2 as an example:

```
fun i() = w + x + y + z
```

The safely linked closure for `i` is shown in in Figure 1; it contains a code pointer plus a pointer to the record containing `w,x,y,z`. Accessing each of these variables inside `i` requires following 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 the available machine registers and cause unnecessary register spilling, but this can be fixed by only keeping a limited number of intermediate variables in the "lazy display" (registers).

To summarize, we used the variable-lifetime information to decide the final closure layout. This is only an optimization to make the closure creation and the access faster. Regardless of what variables we put in the registers and whether or not we use safely linked closures, the actual algorithm would always satisfy the SSC rule as long as we never share closures that bring in new variables (see Section 4.4.3).

For our `iter` example in Figure 7, not only `CR` is shared by all three functions `h`, `J`, and `Q`, but its creation is outside the `h` loop. Thus, each iteration of `h` manages to call two escaping functions without any memory traffic. This is achieved without treating tail recursion as a special case (as was done by Kranz [1987]). Instead, we only used the simple call graph, the variable life-time information, and the closure-strategy analysis, all of which are useful for other purposes anyway. This is an important strength of our new algorithm.

4.5 Remarks

Graph-coloring global register allocation and targeting, which have been implemented by Lal George [1994; 1996], will accomplish most control transfers (function calls) (such as line 12 and 13 in Figure 7) 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 [1990] shows how complicated things become when it's necessary to re-arrange a stack frame.

A 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. A stack implementation that uses non-standard layout can also have a low per-frame overhead, but it still has to reserve per-frame space for *all* local variables used in the function. The technique of associating the return address with a liveness descriptor (see Section 2) only improves the heap-space usage; the dead entries in the frame will not be reclaimed until we pop the frame off the stack. The continuation closure in our scheme tends to be more compact because it only holds free variables that are live in the rest of function body.

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 [Kranz 1987], all tail recursions are assigned 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 3.2, we have shown how our algorithm implements tail recursions very efficiently (i.e., function `iter`). In this section, we use more examples to show how the new algorithm deals with other common control structures such as function

```
fun seq(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
```

Fig. 9. Function `seq` in Standard ML

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

Fig. 10. Function `seq` in CPS

application in sequence, known function call, and general recursion.

5.1 Function calls in sequence

A common control structure in functional programs is to make a sequence of function applications as shown by the example in Figure 9. Here, function `g`—which is a formal parameter of `seq`—is called three times in a row. Both the parameters (i.e., `g,u,v,w`) and the local variables (i.e., `x,y,z`) are placed in registers by default. In a stack-based scheme, an activation record will be pushed onto the stack when function `f` is invoked; then each time before `g` is called, certain variables in registers must be saved to the stack. For example, before the first call to `g`, the registers holding `g` and `w` must be saved so that they can still be retrieved 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 the new technique of using callee-save registers, heap allocation of activation records can be made almost as efficient as stack allocation [Appel and Shao 1996]. The idea is that we can allocate most parts of the current activation record in callee-save registers. With careful lifetime analysis, register save/restore around function calls can often be eliminated or amalgamated, so function calls in sequence need

```
01 fun f(C0,C1,C2,C3,g,u,v,w) =
02   let fun J0(J1,J2,J3,x) =
03         let fun K0(K1,K2,K3,y) =
04               let fun Q0(Q1,Q2,Q3,z) =
05                     let val v = select(4,Q1)
06                         val r = Q3+Q2+z+v+1
07                         val C0 = select(0,Q1)
08                         val C1 = select(1,Q1)
09                         val C2 = select(2,Q1)
10                         val C3 = select(3,Q1)
11                      in C0(C1,C2,C3,r)
12                     end
13                   val g0 = select(0,K2)
14                in g0(K2,Q0,K1,y,K3,y,K2)
15               end
16             val g0 = select(0,J2)
17          in g0(J2,K0,J1,J2,x,x,J3)
18         end
19       val CR = (C0,C1,C2,C3,v)
20       val g0 = select(0,g)
21    in g0(g,J0,CR,g,w,u,v)
22   end
```

Fig. 11. Function seq after closure conversion

Table II. Raw free variables and closure strategies for function seq

Function	**Stage Number**	**Raw Free Variables**	**Closure Strategy**
seq	1	$\emptyset$	1 slot
J	2	$\{(\mathtt{C},4,4),(\mathtt{v},4,4),(\mathtt{g},2,3),(\mathtt{w},2,2)\}$	3 slots
K	3	$\{(\mathtt{C},4,4),(\mathtt{v},4,4),(\mathtt{g},3,3),(\mathtt{x},3,4)\}$	3 slots
Q	4	$\{(\mathtt{C},4,4),(\mathtt{v},4,4),(\mathtt{y},4,4),(\mathtt{x},4,4)\}$	3 slots

only allocate one heap record.

Figures 10 and 11 list the CPS and CLO code for function seq. Table II lists the stage numbers, raw free variables, and closure strategies for function f and continuations J, K, and Q. During the closure conversion of seq (as shown in Figure 11), 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) into 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 is necessary. This is because the lifetimes of w and x (also g and y) do not overlap, so they can just share one callee-save register (i.e., J3 and K3, K2 and Q2).

```
fun map(f,l) =
  let fun m(z) =
        if (z=nil) then nil
        else let val a = car z
                 val r = cdr z
              in (f a)::(m r)
             end
   in m l
  end
```

Fig. 12. Function `map` in Standard ML

5.2 Lambda lifting on known functions

Lambda lifting [Johnsson 1985] 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 efficient code [Kranz 1987]. 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 in the heap or stack (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, that is, 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-analysis algorithm described in Section 4.3 conservatively represents all continuation functions using the same fixed number of callee-save registers. We can relax this restriction when we know the control flow. For example, the continuation parameter of a known user function can be represented in any number of callee-save registers. All we need is to make sure that the actual continuation is represented in the same way. This special calling convention is especially desirable for general recursion such as the `map` function as shown Figure 12.

Figure 13 gives a version of the `map` function written in CPS. Notice that the recursive function `m` is called only at two places: one by function `map` with `C` as

```
fun map(C,f,l) =
  let fun m(J,z) =
        if (z=nil) then J(nil)
        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
```

Fig. 13. Function map in CPS

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 than the first. We can represent all normal continuation functions in three callee-save registers, but represent continuations J and Q in two callee-save registers. Figure 14 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. More generally, if the standard calling convention uses k callee-save registers and the return continuation Q has n free variables, then Q will benefit from using $min(k, n)$ callee-save registers.

6. MEASUREMENT

We have implemented our new closure-conversion algorithm in the Standard ML of New Jersey compiler version 1.03z. In order to find out how much performance gain we can get from our new algorithm, we have measured the performance of six different compilers on ten SML benchmarks.

Table III gives an overview of the ten benchmarks. For each benchmark, we list the size of the source program (in number of lines) and briefly describe its functionality.

The six compilers we use are all simple variations of the SML/NJ compiler ver-

```
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=nil) then J0(J1,J2,nil)
        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
```

Fig. 14. Function map after closure conversion

sion 1.03z. All six compilers satisfy the “safe for space complexity” rule, and all use the type-directed compilation techniques described in a companion paper [Shao and Appel 1995] to allow arguments to be passed in registers and to support more efficient data representations. The “lazy display” technique is implemented in all six compilers, but it is used more effectively in compilers that use the new closure conversion algorithm, because of their more extensive use of shared closures.

sml.occ. This version uses the old closure-conversion algorithm [Appel 1992; Appel and Shao 1992]. More specifically, it uses the linked closure representation if it is space safe, otherwise it uses the flat closure representation. Continuation closures are represented using three callee-save registers.

sml.gp1,sml.gp2,sml.gp3,sml.gp4. These compilers all use the new closure conversion algorithm described in Section 4. They respectively use one, two, three, or four (general-purpose) callee-save registers to represent continuation closures.

sml.fp3. This compiler uses the new closure conversion algorithm described in Section 4. Continuation closures are represented using three general-purpose callee-

Table III. General information about the benchmark programs

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

Table IV. A comparison of execution time

Program	Basis	occ	gp1	gp2	gp3	gp4	fp3
	(secs)	(ratio)	(ratio)	(ratio)	(ratio)	(ratio)	(ratio)
BHut	30.5	1.00	0.77	0.76	0.73	0.75	0.76
Boyer	2.5	1.00	0.92	0.91	0.90	0.90	0.96
Sieve	35.6	1.00	0.84	0.86	0.85	0.86	0.98
KB	7.5	1.00	0.87	0.85	0.81	0.81	0.92
Lexgen	11.5	1.00	0.91	0.89	0.91	0.88	0.93
Yacc	4.4	1.00	1.02	0.97	0.95	0.96	0.98
Life	1.3	1.00	0.91	0.92	0.90	0.89	0.89
Simple	22.2	1.00	0.88	0.84	0.83	0.94	0.86
Ray	20.3	1.00	0.97	0.99	0.98	0.98	0.97
VLIW	16.1	1.00	0.81	0.77	0.76	0.76	0.76
Average		1.00	0.89	0.88	0.86	0.87	0.90

save registers and three floating-point callee-save registers.

All measurements are done on a DEC5000/240 workstation with 128 mega-bytes of memory. In Table IV, we give the execution time of running the benchmarks under the above six compilers. Only the execution time (user time plus garbage collection time plus system time, in seconds) for the **occ** compiler is shown; the performance for other compilers is denoted as a ratio relative to the **occ** compiler. The garbage-collection time (corresponding to the data in Table IV) is listed separately in Table V. Similarly, Table VI, VII, and VIII respectively compare the heap allocation (in mega-bytes), the compilation time (in seconds), and the code size (in kilo-bytes).

In Table IX, we list the number of memory fetches (in millions) for local/global variables and the allocation profile of various kinds of closures for the **occ** and **gp3** compilers. Here, “escape”, “known”, and “cont” are respectively the total size of closures (in mega-words) allocated for escaping user functions, known user functions, and continuation functions; heap allocation for other non-closures is not listed in Table IX (but is included in Table VI).

We can draw the following conclusions from these comparisons:

— The **gp3** compiler has exactly the same setup as the **occ** compiler except that one uses the new closure conversion algorithm, and the other uses the old algorithm.

Table V. A comparison of garbage collection time

Program	**Basis** (secs)	**occ** (ratio)	**gp1** (ratio)	**gp2** (ratio)	**gp3** (ratio)	**gp4** (ratio)	**fp3** (ratio)
BHut	1.61	1.00	0.81	0.85	0.78	0.80	0.83
Boyer	1.03	1.00	0.98	0.97	0.93	0.93	1.01
Sieve	17.9	1.00	0.70	0.72	0.72	0.73	0.85
KB	0.94	1.00	0.98	0.98	0.98	0.99	0.99
Lexgen	0.83	1.00	0.90	0.89	0.83	0.88	0.94
Yacc	1.05	1.00	1.12	1.02	1.04	1.04	1.07
Life	0.02	1.00	1.00	1.50	0.00	1.00	1.00
Simple	3.43	1.00	0.90	0.89	0.90	0.91	0.90
Ray	0.05	1.00	1.80	1.60	2.20	2.00	1.80
VLIW	0.53	1.00	0.94	0.83	0.96	0.94	1.06

Table VI. A comparison of total heap allocation

Program	**Basis** (MB)	**occ** (ratio)	**gp1** (ratio)	**gp2** (ratio)	**gp3** (ratio)	**gp4** (ratio)	**fp3** (ratio)
BHut	619.0	1.00	0.55	0.50	0.44	0.44	0.45
Boyer	30.8	1.00	0.86	0.80	0.75	0.73	0.88
Sieve	253.9	1.00	0.72	0.72	0.70	0.70	0.83
KB	156.7	1.00	0.79	0.70	0.59	0.57	0.79
Lexgen	96.9	1.00	0.78	0.71	0.64	0.49	0.69
Yacc	57.5	1.00	0.89	0.81	0.77	0.75	0.78
Life	10.2	1.00	0.77	0.73	0.68	0.63	0.68
Simple	323.5	1.00	0.82	0.68	0.64	0.70	0.57
Ray	331.6	1.00	0.92	0.96	0.93	0.92	0.93
VLIW	160.8	1.00	0.80	0.75	0.70	0.71	0.71
Average		1.00	0.79	0.74	0.68	0.66	0.73

On average, the **gp3** compiler reduces heap allocation by 32% (closure allocation by 40%) and memory fetches for local/global variables by 46%; and improves the already efficient code generated by the **occ** compiler by 14%. The **gp3** compiler also uniformly generates more compact code, achieving an average of 19% reduction in code size over the **occ** compiler. **BHut** and **VLIW** achieve up to respectively 27% and 24% speedup in execution time, because they get significant benefits from using safely linked closures.

— Varying the number of callee-save registers under the new closure conversion algorithm has little effect on the execution time (within 6% range, 3% on average), but has a large effect on the total heap allocation. The **gp3** compiler is about 3-4% faster than the **gp1** compiler, but its total heap allocation is more than 11% less.

— The effect of the new closure algorithm on the garbage collection time (g.c. time, see Table V) varies dramatically depending on the benchmarks. Eight of the ten benchmarks we measured spend less time in garbage collection (because of less heap allocation), however, the g.c. time for **Ray** is almost doubled. This is not surprising since the new closure algorithm has completely different allocation behavior from the old algorithm. Having more callee-save registers (especially the **fp3** compiler) generally increases the g.c. time, which somewhat reflects the heap

Table VII. A comparison of compilation time

Program	Basis (secs)	occ (ratio)	gp1 (ratio)	gp2 (ratio)	gp3 (ratio)	gp4 (ratio)	fp3 (ratio)
BHut	59.1	1.00	0.99	0.92	0.96	0.95	1.01
Boyer	26.4	1.00	0.99	0.96	0.96	0.98	1.01
Sieve	31.8	1.00	0.99	0.98	0.97	0.99	1.02
KB	19.4	1.00	1.03	0.97	0.98	1.01	1.03
Lexgen	37.8	1.00	0.91	0.92	0.86	0.89	0.99
Yacc	132.2	1.00	0.98	0.93	0.95	0.95	1.01
Life	4.8	1.00	1.04	1.01	0.98	1.02	1.04
Simple	64.6	1.00	0.85	0.83	0.82	0.78	0.86
Ray	15.5	1.00	1.01	0.98	0.97	0.98	1.06
VLIW	185.9	1.00	0.98	0.90	0.91	0.97	0.98
Average		1.00	0.98	0.94	0.94	0.95	1.00

Table VIII. A comparison of code size

Program	Basis (KB)	occ (ratio)	gp1 (ratio)	gp2 (ratio)	gp3 (ratio)	gp4 (ratio)	fp3 (ratio)
BHut	103.1	1.00	0.83	0.81	0.81	0.76	0.83
Boyer	107.2	1.00	0.93	0.92	0.92	0.93	0.92
Sieve	74.1	1.00	0.85	0.83	0.83	0.84	0.87
KB	47.1	1.00	0.91	0.86	0.86	0.88	0.88
Lexgen	105.5	1.00	0.83	0.80	0.79	0.79	0.80
Yacc	418.2	1.00	0.81	0.76	0.75	0.75	0.75
Life	14.5	1.00	0.91	0.88	0.88	0.87	0.88
Simple	183.1	1.00	0.64	0.61	0.59	0.56	0.58
Ray	53.5	1.00	0.87	0.84	0.85	0.85	0.87
VLIW	426.9	1.00	0.82	0.78	0.78	0.78	0.78
Average		1.00	0.84	0.81	0.81	0.80	0.82

allocation data in Table VI.

— From the allocation profiling data in Table IX, we can see that most of the reduction in heap allocation is from the continuation closures. On some benchmarks, the new strategy builds more known-function closures because it now does controlled lambda-lifting (see Section 4.3 and 5.2). For example, for function `iter` in Figure 7, the new strategy manages to eliminate all continuation closures (inside the loop) by building one closure for known function `h` (outside the loop).

— The new closure-conversion algorithm surprisingly improves the compilation time by nearly 6%. This is probably because the old algorithm used in **occ** compiler contains expensive ad-hoc heuristics, while the new algorithm is much more systematic. Another reason might be that the new algorithm generates less code, thus requires less instruction scheduling time.

— Using three floating-point callee-save registers (i.e., the **fp3** compiler) does not achieve any better performance than using no floating-point callee-save registers. The slowdown mostly comes from benchmarks such as **Sieve** and **KB** that frequently use first-class continuations and exception handlers. First-class continuations and exceptions may be put into a record or stored into some reference cell, so

Table IX. Breakdown of closure access and allocation

	Closure Access			**Closure Allocation**		
	(mem-read in millions)			(escape+known+cont in mega-words)		
Program	**occ**	**gp3**	**saving**	**occ**	**gp3**	**saving**
BHut	87.6	37.1	57.7%	0.22+0.18+76.0	0.12+1.59+36.6	49.91%
Boyer	4.58	2.58	43.6%	1.91+4.88+0.95	1.18+0.48+3.18	37.56%
Sieve	47.2	27.5	41.8%	7.28+11.4+26.5	7.28+8.29+9.84	43.71%
KB	16.0	13.0	18.7%	12.5+0.04+24.2	5.99+1.07+12.9	45.57%
Lexgen	15.9	7.57	52.4%	1.47+0.18+17.1	0.48+0.67+6.90	56.92%
Yacc	9.75	4.21	56.8%	0.10+2.71+8.03	0.09+2.07+5.33	30.89%
Life	1.57	0.66	58.0%	0.06+0.00+1.50	0.06+0.06+0.73	46.05%
Simple	76.5	45.8	40.1%	3.47+0.55+62.4	2.64+4.13+35.5	36.45%
Ray	20.2	13.6	32.5%	0.00+0.01+15.2	0.00+2.11+11.5	10.35%
VLIW	40.7	16.8	58.7%	7.38+2.60+33.5	6.78+3.07+15.8	40.89%
Average			46.0%			39.83%

they must be representable in just one word, not as k separate callee-save registers; when a continuation is captured, the k-register representation has to be packaged into a single word by making a record on the heap; when a continuation is triggered (by `throw`), the single-word representation must be unpackaged into k callee-save registers. This overhead is higher if more callee-save registers are used.

7. RELATED WORK

Rabbit [Steele 1978] is the first compiler that uses the continuation-passing style as the intermediate language; it is also the first compiler that represents the stack frames as continuation closures. Rozas's Liar compiler [Rozas 1984] used closure analysis to choose specialized representations for different kinds of closures; Kranz's Orbit compiler [Kranz et al. 1986; Kranz 1987] uses six different closure-allocation strategies for different kinds of functions; Appel and Jim [1988] investigated closure-sharing strategies and proposed many alternative closure representations. Unfortunately, all these closure-analysis techniques violate the "safe for space complexity" (SSC) rule due to unsafe closure sharing. The closure-conversion algorithm described in this paper combines all of these analyses (except stack allocation) and more, while still satisfying the SSC rule.

Hanson [1990] showed the complexity of implementing tail calls efficiently on a conventional stack. In our heap-based scheme, the correctness is straightforward because dead frames are automatically reclaimed by the garbage collector; the efficiency is achieved by using the loop-header technique [Appel 1994] to hoist the loop-invariant free variables out of the tail recursion, and using the callee-save registers [Appel and Shao 1992] to simulate the top reusable stack frames.

Johnsson [1985] invented lambda lifting where free variables of a known function can be treated as extra arguments (thus allocated in registers). Wand and Steckler [1994] proposed the idea of light-weight closures which leave out certain free variables if they are accessible from other places.

Some compilers [Steele 1978; Kranz et al. 1986; Cardelli 1984] perform closure conversion and closure analyses as part of their translation from lambda calculus or continuation-passing style into machine code. But it is useful to separate the closure introduction from machine code generation so that the compiler is more modular; this has been done in compilers based on ordinary λ-calculus (through lambda lifting) [Cousineau et al. 1985; Johnsson 1985] and on continuation-passing style (using closure-passing style) [Appel and Jim 1989; Kelsey and Hudak 1989].

Both Chow [1988] and Steele and Sussman [1980] observed that data-flow analysis can help decide whether to put variables in caller-save or callee-save registers. We are the first to show how to represent callee-save registers in continuation-passing style [Appel and Shao 1992; Appel 1992] and how to use compile-time variable-lifetime information to do a much better job of it (this paper).

Local variables of different functions with nonoverlapping live ranges can be allocated to the same register or global without any save/restore [Chow 1988; Gomard and Sestoft 1991]. We achieve this by allocating part of the closures (for known functions and continuations) in both caller- and callee-save registers, and then using the graph-coloring global register allocation and targeting algorithms [Chaitin 1982; Briggs et al. 1989; George et al. 1994; George and Appel 1996].

Recently, Clinger [1998] gave a formal and implementation-independent definition of the proper tail recursion and the SSC rule. Our current paper argues for the importance of the SSC rule (in implementing functional languages) and presents a closure-allocation scheme that is both efficient and safe for space.

8. CONCLUSIONS

We have presented an efficient and safe-for-space closure-conversion algorithm that integrates and improves previous closure-analysis techniques. Our algorithm uses simple and effective heuristics (based on compile-time control-flow and variable life-time information) to exploit callee-save registers and construct safely linked closures. Unlike any previous schemes, our algorithm satisfies the "safe for space complexity" scoping rule while still supporting very efficient closure allocation and variable access.

Our measurement shows that the new closure-conversion algorithm is a great success. The closure-conversion algorithm itself is faster than our previous algorithm (see Table VII). It makes programs smaller (by an average of 19%) and faster (by an average of 14%). It decreases the rate of heap allocation by 32%, and by obeying the "safe for space complexity" rule and keeping closures small, it helps reduce the amount of live data preserved by garbage collection.

The closure-analysis techniques presented in this paper can also be applied to compilers that do not use CPS as their intermediate language. Both safely linked closures and good use of callee-save registers are essential in building high-quality compilers that can generate efficient code while still satisfying the "safe for space complexity" rule.

9. AVAILABILITY

The new closure-conversion algorithm presented in this paper is now implemented and released with the Standard ML of New Jersey (SML/NJ) compiler and the FLINT/ML compiler [Shao 1997]. SML/NJ is a joint work by Lucent, Princeton, and Yale. FLINT is a modern compiler infrastructure developed at Yale University. Both FLINT and SML/NJ are available at the following web site:

`http://flint.cs.yale.edu`

ACKNOWLEDGEMENTS

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

REFERENCES

Appel, A. W. 1987. Garbage collection can be faster than stack allocation. *Information Processing Letter 25,* 4, 275–279.

Appel, A. W. 1992. *Compiling with Continuations.* Cambridge University Press, New York.

Appel, A. W. 1994. Loop headers in λ-calculus or CPS. *Lisp and Symbolic Computation 7,* 337–343. Also available as Princeton University Tech Report CS-TR-460-94.

Appel, A. W. and Jim, T. 1988. Optimizing closure environment representations. Tech. Rep. 168, Dept. of Computer Science, Princeton University, Princeton, NJ.

Appel, A. W. and Jim, T. 1989. Continuation-passing, closure-passing style. In *Sixteenth ACM Symp. on Principles of Programming Languages.* ACM Press, New York, 293–302.

Appel, A. W. and MacQueen, D. B. 1991. Standard ML of New Jersey. In *Third Int'l Symp. on Prog. Lang. Implementation and Logic Programming*, M. Wirsing, Ed. Springer-Verlag, New York, 1–13.

Appel, A. W. and Shao, Z. 1992. Callee-save registers in continuation-passing style. *Lisp and Symbolic Computation 5,* 3, 191–221.

Appel, A. W. and Shao, Z. 1996. Empirical and analytic study of stack versus heap cost for languages with closures. *Journal of Functional Programming 6,* 1 (January), 47–74.

Augustsson, L. 1989. Garbage collection in the $< \nu, g >$-machine. Tech. Rep. PMG memo 73, Dept. of Computer Sciences, Chalmers University of Technology, Goteborg, Sweden. December.

Ball, T. and Larus, J. R. 1993. Branch prediction for free. In *Proc. ACM SIGPLAN '93 Conf. on Prog. Lang. Design and Implementation.* ACM Press, New York, 300–313.

Briggs, P., Cooper, K. D., Kennedy, K., and Torczon, L. 1989. Coloring heuristics for register allocation. In *Proc. ACM SIGPLAN '89 Conf. on Prog. Lang. Design and Implementation.* ACM Press, New York, 275–284.

Cardelli, L. 1984. Compiling a functional language. In *Proc. of the 1984 ACM Conference on Lisp and Functional Programming.* ACM Press, New York, 208–217.

Chaitin, G. J. 1982. Register allocation and spilling via graph coloring. In *Symposium on Compiler Construction.* ACM Sigplan, New York, 98–105.

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

Cheng, P., Harper, R., and Lee, P. 1998. Generational stack collection and profile-driven pretenuring. In *Proc. ACM SIGPLAN '98 Conf. on Prog. Lang. Design and Implementation.* ACM Press, New York, 162–173.

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

Clinger, W. D. 1998. Proper tail recursion and space efficiency. In *Proc. ACM SIGPLAN '98 Conf. on Prog. Lang. Design and Implementation.* ACM Press, New York, 174–185.

Clinger, W. D., Hartheimer, A. H., and Ost, E. M. 1988. Implementation strategies for continuations. In *1988 ACM Conference on Lisp and Functional Programming.* ACM Press, New York, 124–131.

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

Diwan, A., Tarditi, D., and Moss, E. 1994. Memory subsystem performance of programs using copying garbage collection. In *Proc. 21st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages.* ACM Press, New York, 1–14.

Flanagan, C., Sabry, A., Duba, B. F., and Felleisen, M. 1993. The essence of compiling with continuations. In *Proc. ACM SIGPLAN '93 Conf. on Prog. Lang. Design and Implementation.* ACM Press, New York, 237–247.

George, L. and Appel, A. W. 1996. Iterated register coalescing. In *Proc. Twenty-third Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. ACM Press, New York, 208–218.

George, L., Guillaume, F., and Reppy, J. 1994. A portable and optimizing backend for the SML/NJ compiler. In *Proceedings of the 1994 International Conference on Compiler Construction*. Springer-Verlag, New York, 83–97.

Gomard, C. K. and Sestoft, P. 1991. Globalization and live variables. In *Proceedings of the 1991 Symposium on Partial Evaluation and Semantics-Based Program Manipulation*. ACM Press, New York, 166–177.

Hanson, C. 1990. Efficient stack allocation for tail-recursive languages. In *1990 ACM Conference on Lisp and Functional Programming*. ACM Press, New York, 106–118.

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

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

Jouppi, N. P. 1993. Cache write policies and performance. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. ACM Press, New York, 191–201.

Kelsey, R. and Hudak, P. 1989. Realistic compilation by program transformation. In *Sixteenth ACM Symp. on Principles of Programming Languages*. ACM Press, New York, 281–292.

Kranz, D. 1987. ORBIT: An optimizing compiler for Scheme. Ph.D. thesis, Yale University, New Haven, CT.

Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., and Adams, N. 1986. ORBIT: An optimizing compiler for Scheme. *SIGPLAN Notices (Proc. Sigplan '86 Symp. on Compiler Construction) 21,* 7 (July), 219–233.

Landin, P. J. 1964. The mechanical evaluation of expressions. *Computer Journal 6,* 4, 308–320.

Milner, R., Tofte, M., and Harper, R. 1990. *The Definition of Standard ML*. MIT Press, Cambridge, Massachusetts.

Morrisett, G., Crary, K., Glew, N., and Walker, D. 1998. Stack-based typed assembly language. In *Proc. 1998 International Workshop on Types in Compilation: LNCS Vol 1473*, X. Leroy and A. Ohori, Eds. Springer-Verlag, Kyoto, Japan, 28–52.

Muchnick, S. S. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, San Francisco, California.

Ramsey, N. and Peyton Jones, S. 1999. A single intermediate language that supports multiple implementations of exceptions. Technical paper yet to be published.

Reppy, J. H. 1993. A high-performance garbage collector for Standard ML. Technical memorandum, AT&T Bell Laboratories, Murray Hill, NJ. January.

Rozas, G. J. 1984. Liar, an Algol-like compiler for scheme. S.B. thesis, MIT Dept. of Computer Science and Electrical Engineering.

Ryder, B. G. and Paull, M. C. 1986. Elimination algorithms for data flow analysis. *ACM Computing Surveys 18,* 3 (September), 277–316.

Shao, Z. 1994. Compiling Standard ML for efficient execution on modern machines. Ph.D. thesis, Princeton University, Princeton, NJ. Tech Report CS-TR-475-94.

Shao, Z. 1997. Flexible representation analysis. In *Proc. 1997 ACM SIGPLAN International Conference on Functional Programming (ICFP'97)*. ACM Press, New York, 85–98.

Shao, Z. and Appel, A. W. 1994. Space-efficient closure representations. In *1994 ACM Conference on Lisp and Functional Programming*. ACM Press, New York, 150–161.

Shao, Z. and Appel, A. W. 1995. A type-based compiler for Standard ML. In *Proc. ACM SIGPLAN '95 Conf. on Prog. Lang. Design and Implementation*. ACM Press, New York, 116–129.

Shivers, O. 1991. Control-flow analysis of higher-order languages. Ph.D. thesis, Carnegie Mellon Univ., Pittsburgh, Pennsylvania. CMU-CS-91-145.

STEELE, G. L. 1978. Rabbit: a compiler for Scheme. Tech. Rep. AI-TR-474, MIT, Cambridge, MA.

STEELE, G. L. AND SUSSMAN, G. J. 1980. The dream of a lifetime: A lazy variable extent mechanism. In *Proceedings of the 1980 LISP Conference.* ACM Press, Stanford, 163–172.

TARJAN, R. E. 1974. Testing flow graph reducibility. *Journal of Computer and System Science 9,* 3 (December), 355–365.

UNGAR, D. M. 1986. *The Design and Evaluation of A High Performance Smalltalk System.* MIT Press, Cambridge,MA.

WAND, M. AND STECKLER, P. 1994. Selective and lightweight closure conversion. In *Proc. 21st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages.* ACM Press, New York, 435–445.

Received September 1998; revised October 1999; accepted December 1999