LISP AND SYMBOLIC COMPUTATION: An International Journal, 5, 191–221, 1992

Callee-save Registers in Continuation-passing Style

ANDREW W. APPEL* (*appel@princeton.edu*)

ZHONG SHAO† (*zsh@princeton.edu*)

Department of Computer Science, Princeton University, Princeton, NJ 08544-2087

Keywords: Register Allocation, Continuation-passing Style, Procedure Call

Abstract. Continuation-passing style (CPS) is a good abstract representation to use for compilation and optimization: it has a clean semantics and is easily manipulated. We examine how CPS expresses the saving and restoring of registers in source-language procedure calls. In most CPS-based compilers, the context of the calling procedure is saved in a "continuation closure"—a single variable that is passed as an argument to the function being called. This closure is a record containing bindings of all the free variables of the continuation; that is, registers that hold values needed by the caller "after the call" are written to memory in the closure, and fetched back after the call.

Consider the procedure-call mechanisms used by conventional compilers. In particular, registers holding values needed after the call must be saved and later restored. The responsibility for saving registers can lie with the caller (a "caller-saves" convention) or with the called function ("callee-saves"). In practice, to optimize memory traffic, compilers find it useful to have some caller-saves registers and some callee-saves.

"Conventional" CPS-based compilers that pass a pointer to a record containing all the variables needed after the call (i.e., the continuation closure), are using a caller-saves convention. We explain how to express callee-save registers in Continuation-Passing Style, and give measurements showing the resulting improvement in execution time.

1. Introduction

A compiler that uses a good abstract representation (like λ-calculus) as a basis for compilation can perform program transformations and optimizations with a free hand; the clean semantics of the abstract representation frees the optimizer from too much worry about the technical details of storage allocation, etc. On the other hand, real compilers must deal with all the nitty-gritty of modern machines: a fixed number of fast registers instead of an unbounded name space; the necessity to save registers when changing environments at a procedure call, and so on. It is not always easy to reconcile these different aspects of compilation.

*Supported in part by NSF Grant CCR-9002786.

†Supported in part by NSF Grant CCR-8914570.

What can easily happen is that the "abstract" optimizer will be unaware of low-level representation decisions, so that it loses the chance to optimize them; but the "concrete" optimizer that decides register-allocation issues will be bound by rigid conventions and bad decisions of the abstract optimizer.

A good example of a semantically clean abstract language that makes many kinds of high-level optimizations easy is Continuation-Passing Style (CPS). As first used in compiling by Steele[20], CPS was largely divorced from low-level representations: its target machine was a Lisp compiler. The ORBIT compiler of Kranz *et al*[17] is organized so that the variables of the CPS language correspond directly to registers or to memory locations (fields of closures), and the optimizer can tell which is which, enabling more intelligent decisions about program transformations. The transformational compiler of Kelsey and Hudak[15], and the Standard ML of New Jersey compiler[4, 3], refine this notion further so that the "abstract" optimizer can control representation decisions while still staying within the original, semantically clean and powerful Continuation Passing Style.

Our claim is that the functional style of CPS can simply and powerfully express the low-level details of register allocation, saving and restoring of registers at procedure calls, and parameter-passing conventions. Optimizations and conventions used by "conventional" compilers should also be usable by "continuation-passing" compilers *without* making the CPS language ugly or more complex.

In this paper we demonstrate that optimization of register saving and restoring at procedure calls can be expressed within Steele's original CPS language; this improves the already-efficient code generated by the Standard ML of New Jersey compiler by about 6.3% on a MIPS and 14.4% on a SPARC.

2. Callee-save registers

A conventional compiler on a conventional machine will often use registers to hold local variables and temporaries (e.g. internal nodes of expression trees). When one procedure calls another, both procedures may use the same registers for different purposes, and there must be some convention for saving the registers and restoring them. For example, it could be that the calling procedure (the "caller") must assume that the procedure it calls (the "callee") may put any values in registers without preserving the original values; therefore the caller must save registers (copy them into memory) before the call, and restore them (copy them from memory) after the call. This is a "caller-save" convention. On the other hand, it could be that the callee is responsible for leaving registers exactly as it found them;

therefore, if the callee wants to use some registers for temporary values, it must save them prior to use and restore them before returning. This is a "callee-save" convention.

Which approach is preferable? Consider temporaries that hold the internal nodes of expression-tree evaluation. Since most procedure calls occur outside of expression trees, the registers that hold these temporaries are "idle" at the time of a procedure call—the caller does not care if their contents are lost. Therefore, a caller-save convention is best for such registers.

On the other hand, consider a small "leaf" procedure—one that does not call other procedures, and which uses very few registers before returning. If the caller wanted to preserve several local (register) variables across the call, then under a caller-save convention they would have to be written to memory before the call and read from memory afterwards, even though the leaf procedure does not modify them. In this case, a callee-save convention seems best.

In practice, compiler-writers have opted to have some of each. The Berkeley C compiler for the VAX, for example, uses 6 caller-save and 6 callee-save registers (in addition to 4 special-purpose registers like the stack pointer, frame pointer, etc.). With a mixed caller/callee-save convention like this, variables not needed after the call can be put in caller-save registers and not saved at all; leaf procedures can use caller-save registers to the extent possible. Though the use of caller- and callee-save registers seems to be a very old part of lore of compilers, it appears that only recently have compilers attempted to optimize the placement of variables into caller- and callee-save registers[9].

3. Continuation-passing style

Continuation-passing style (CPS) is a language similar to 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—there is no useful distinction between normal-order and call-by-value, for example.

For the purposes of this paper, we will express CPS using ML notation, albeit severely constrained—see figure 1. An atom A can be a variable or a constant; a *record* can be constructed out of a sequence $(A, A, \ldots, A)$ of atoms. If an n-element record is bound to a variable v using `let val` $v = (A_0, \ldots, A_{n-1})$ `in` E `end`, then anywhere in the subexpression E the i^{th} field may be fetched using `select(`i, v`)`; but (as the syntax shows) the result must be bound to a variable, not used as an arbitrary subexpression. The

$$
\begin{array}{lcl}
V & \rightarrow & \textit{variable} \\
I & \rightarrow & \textit{integer constant} \\
P & \rightarrow & \textit{arithmetic operator} \\
A & \rightarrow & V \\
A & \rightarrow & I \\
\\
F & \rightarrow & V(V, V, \ldots, V) \ = \ E \\
F & \rightarrow & F \ \texttt{and} \ F \\
\\
E & \rightarrow & \texttt{let val} \ V \ = \ (A, A, \ldots, A) \ \texttt{in} \ E \ \texttt{end} \\
E & \rightarrow & \texttt{let val} \ V \ = \ \texttt{select}(I, A) \ \texttt{in} \ E \ \texttt{end} \\
E & \rightarrow & \texttt{let val} \ V \ = \ \texttt{offset}(I, A) \ \texttt{in} \ E \ \texttt{end} \\
E & \rightarrow & \texttt{let val} \ V \ = \ P(A, \ldots, A) \ \texttt{in} \ E \ \texttt{end} \\
E & \rightarrow & \texttt{let fun} \ F \ \texttt{in} \ E \ \texttt{end} \\
E & \rightarrow & A(A, A, \ldots, A)
\end{array}
$$

Figure 1: Abstract syntax of CPS

syntax for building records, selecting fields, applying primitive arithmetic operators, and defining mutually recursive functions (`fun`) must specify a continuation expression E that will use the result.

On the other hand, function application (shown in the last line of figure 1) does not specify a continuation expression—functions never *return* in the conventional sense. Instead, it is expected that many functions will pass a *continuation function* as one of their arguments. This function can be defined in the ordinary way (by `fun`), and will presumably be invoked by the callee in order to continue the computation.

Variables in ML (and in this continuation-passing style) are pure values; they cannot be updated by side-effect operations. Side effects in ML (and in this CPS) are possible on the mutable store, which we will not discuss here.

The `offset` operator performs simple pointer arithmetic (for use with shared closures, as will be explained below): `val` w = `offset`(i, v) binds w to a record similar to the record value v, except that the j^{th} field of w is the same as the $(i+j)^{\text{th}}$ field of v.

Continuation-passing style has several advantages as an intermediate representation for an optimizing compiler [20, 16, 4]. Because it has simple static scope, in-line expansion of functions (β-reduction) is very simple to express as are constant-folding and other partial evaluations, common-subexpression-elimination, dead variable elimination, loop optimizations, and function calling-sequence specialization.

An example

```
 . . .
fun h(x) = x * w

fun f(g,y) = g(y) + h(z)

val i = (f,1)

let fun m(n) = n+t
    val p = f(m,1)
 in ...p+m(e)...
end
 . . .
```

Figure 2: An example : ML source code

Consider the program fragments in figure 2. After translation into CPS,

```
 . . .
fun h(x,d) =
 let val t1 = x * w
  in d(t1)
 end

fun f(arg,c) =
 let val g = select(arg,0)
     val y = select(arg,1)
     fun j(a) =
         let fun k(b) =
               let val t2 = a+b
                in c(t2)
               end
          in h(z,k)
         end
  in g(y,j)
 end

val i = (f,1)

let fun m(n,r) =
      let val q = n+t
       in r(q)
      end
    fun s(p) = ...p...m...e...
    val u = (m,1)
 in f(u,s)
end
 . . .
```

Figure 3: An example: CPS code

we have the code in figure 3 (the detailed translation rules are described in Steele [20] or Appel [3]).

There are two kinds of functions in our CPS code. *Continuation functions* are introduced in the CPS conversion phase, e.g. `j`, `k`, `s`. We call all non-continuation functions *user functions*. *Continuation variables* are all those formal parameters introduced in CPS-conversion to hold continuation functions: `d`, `c`, `j`, `k`, `r` and `s`. Functions such as `f`,`j`,`k` and `m` are called *escaping functions* if they are passed as arguments or stored in data structures so that the compiler can't identify all the places where they are called. All functions that do not escape are called *known functions*, e.g. `h`. We can do extensive optimizations on known functions since we know all their call sites at compile time.

The CPS code always satisfies the following properties:

- All escaping user functions have two arguments: one is the standard argument (note that an n-tuple is considered to be one argument, as in the call to `f`) and the another is the continuation argument.
- All escaping continuation functions have one argument.
- Known functions may have an arbitrary number of arguments after calling-sequence specialization [4, 3].

4. Closure representations

Continuation-passing style is meant to approximate the operation of a von Neumann computer; each operator of the former corresponds to one (or at most a few) instructions of the latter. Selecting the i^{th} field of a record in the CPS is like a fetch with constant offset on a 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 the CPS is almost the same: the structure of CPS expressions is that a function application is the last thing a function does; the result of a function application is always the result of its parent expression. Thus the function application is also a "jump with arguments." If a "return" from a procedure (in the usual sense) is desired, then a *continuation function* must be made: one of the arguments to the called function will itself be a function c; the called function is expected to call c with its result.

However, the function definitions of continuation-passing style are a bit more powerful than those of conventional computers. Function definitions in CPS have nested static scope; if the function f is statically nested inside the function g, then f can refer to the variables of g. We say that these are *free variables* of f. The notion of a function as a machine-code address does not provide for free variables.

The usual solution to this problem is to represent functions as *closures* [18]. A function with free variables is said to be *open*; a *closure* is a data structure containing both the machine code address of an open function, and bindings for all the free variables of that function. The machine-code implementation of the function knows to find the values of free variables in the closure data structure.

Conventional compilers use an *activation record* to hold the values of variables accessible by the function body. In most CPS-based compilers [17, 15, 4], function parameters and local variables are held in registers, and free variables are accessed from the closure, so that the closure and registers together serve in place of the activation record.

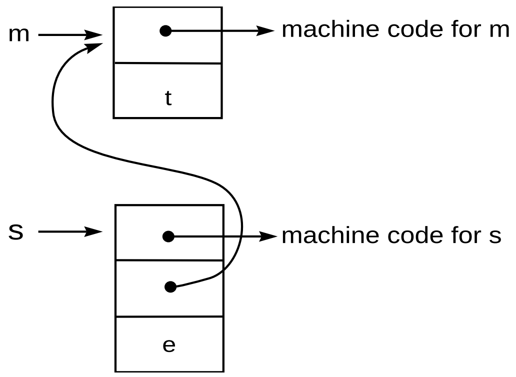
Most CPS-based compilers [20, 17] allocate certain closures on a stack if their pattern of usage can be sufficiently analyzed at compile time, so that they can be efficiently and promptly deallocated. However, using a stack complicates the garbage-collector interface[6], and greatly complicates the efficient implementation of *call-with-current-continuation* [13]. The techniques we present here neither require nor preclude the use of a stack.

We will avoid any technique that calls for side effects to existing closures. In the first place, this corrupts the notion of CPS as a functional, not imperative, notation for program optimization. Second, in a system with garbage collection heap records should be initialized when created (otherwise the collector will trace bad pointers), and then never again stored into (otherwise a generational collector may miss references from old generations into new). Exceptions to this rule can be made only at the expense of significantly complicating the interface to the collector. Using closures (which are created all at once and then not modified) instead of activation records (which are updated piecemeal as functions execute) makes for a clean model of memory access.

Let's return to the example in figure 3. The function m is passed as an argument to f; the function k is passed to h, and so on. Each of these functions has free variables (t is a free variable of m, a and c are free variables of k). The implementation must represent k using some data structure that contains the machine code for computing $a + b$ and also contains the values for a and c.

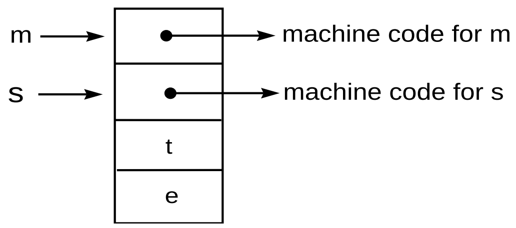
When g is called from within f, the machine-code pointer must be extracted from the closure, arguments y and j must be put in registers, and the jump (to m) must be made. But m must also be able to access its closure; so it is important that the closure-pointer g (which is really m) be placed in a standard register before the jump—in effect, the closure-pointer is an extra argument to the function. Then m can extract free variables through this pointer. The format of the free variables in the closure need not be standardized: the creator of the closure (at `fun m(n,r)...`) and the code that extracts free variables (at `n+t`) need to know the format, but the caller (at `g(y,j)`) need not know where they are, or indeed how many there are. However, the location of the code-pointer within the closure must be standardized so that the caller can find it and jump to it. A typical representation puts the code-pointer at field 0 of the closure-record, and the free variables at other offsets from the closure pointer, perhaps in a linked list of closures.

One interesting trick [17, 5] is to let several functions share a single closure. The functions m and s might ordinarily be represented like this:



The closure for m has the value for the free variable t; the closure for s has free variables m and e.

Now consider this data structure:



The value s is really a pointer to the middle of the closure record! The caller of s does not know this, however, and when it extracts what it thinks is field 0, it gets (correctly) the code-pointer for s. Of course, the machine code functions m and s must know the closure format; in particular, s derives the closure-value m by subtracting from its own closure-pointer, without a fetch! (This is the purpose of the `offset` primitive of the CPS.) Since there is some overhead for record creation—creating a record of size n costs $Bn + C$ operations, for some B and C—the closure-sharing trick is quite useful; in this case there is a savings of $B + C$ per instance of m/s.

Notice that h is never passed as an argument; it appears in the expression $h(z, k)$ in *function position*. Since we know all the call sites, we can choose the representation of h more freely; for example, we could require all the callers to pass the free variable w as an extra argument, so that no closure at all is required.

In general, *escaping functions* must be compiled using a standard closure mechanism; *known functions* can use cheaper, more specialized representations [20], typically with free variables passed as arguments.

5. Closure-passing style

Some compilers[20, 8, 17] perform these 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; we can rewrite the program into an equivalent one in which no function has free variables. This has been done in compilers based on ordinary λ-calculus [14, 10] and on continuation-passing style [15, 4].

We will represent closure creation and use in the continuation-passing style itself; each closure will be an explicit record of the CPS. Our example—as transformed into closure-passing style—is shown in figure 4

The motivation for this transformation is that *all variables in the CPS after closure conversion stand for either registers or constants.* We can continue to rely on the λ-calculus-like semantics and scoping rules of CPS while manipulating a machine-level program.

After closure conversion, the function k is now a closure-record; k' is a function without free variables, which can thus be represented as just a (constant) machine-code pointer; the formal parameter k'' will be bound to the closure k by any caller of k'.

The function h is known, so does not need a closure; however, its free variable w must be passed as an extra argument w'; this in turn means that w is a free variable of f. If f were known, w would become an extra argument; but since f escapes, w goes into its closure.

The original function f made two subroutine calls, to g and h. Neither of these was a tail-call, so each call requires a continuation to be made, respectively j and k. Each continuation requires a closure. Thus the execution of f requires the construction of two closures of size 3 and 4, for a total cost

```
fun h(x,d,w') = let val t1 = x*w'
                    val d' = select(d,0)
                 in d'(d,t1)
                end

fun f'(f'',arg,c) =
     let val g = select(arg,0)
         val y = select(arg,1)

         fun j'(j'',a)=
           let fun k'(k'',b)=
                 let val a = select(k'',1)
                     val v = a+b
                     val c'' = select(k'',2)
                     val c'= select(c'',0)
                  in c'(c'',v)
                 end

                val c'''= select(j'',1)
                val k = (k',a,c''')
                val z' = select(j'',2)
                val w'' = select(j'',3)
             in h(z',k,w'')
            end

         val w = select(f'',2)
         val z = select(f'',1)
         val j = (j',c,z,w)
         val g' = select(g,0)
      in g'(g,y,j)
     end

val f = (f',z,w)
val i = (f,1)

let fun m'(m'',n,r) =
       let val t = select(m'',2)
           val q = n+t
           val r' = select(r,0)
        in r'(r,q)
       end
    fun s'(s'',p) =
       let val e = select(s'',2)
           val m = offset(s'',-1)
        in ...p...m...e...
       end
    val m = (m',s',t,e)
    val s = offset(m,1)
    val u = (m,1)
 in f'(f,u,s)
end
```

Figure 4: Closure-passing style

of $7B + 2C$. It would be nice if j and k could use the closure-sharing trick, but they cannot: the variable a is free in k but bound by j, so it cannot yet exist when the closure for j is made but must exist when k is created.

6. Callee-save registers in CPS

In a sense, when f calls g it saves all the registers it might need after the call in the closure j. This is clearly like a caller-save convention. It might be nice to have some callee-save registers where values could be kept; the trick is to express this kind of convention in continuation-passing style.

6.1 Basic ideas

What we will do is to give every function n extra arguments. We will require that each "user" (non-continuation) function f must pass these arguments to its continuation c, when f (or some function that f calls) eventually calls c. Thus, these extra arguments will behave like callee-save registers.

Another way to look at our new callee-save convention is that each user function will be passed a continuation, as before, but now a continuation is represented using $n+1$ actual parameters. One of these parameters will be the code pointer; the others will be free variables of the continuation. Let us consider the function f from our previous example, letting $n = 3$:

```
fun f'(f'',arg,c0,c1,c2,c3) =
     let . . .
                        c0(t2,c1,c2,c3)
     end
```

Instead of a single continuation-argument c, now f gets four arguments c_0, c_1, c_2, c_3, of which c_0 is the machine-code pointer. When f eventually returns (actually, it is k that calls the continuation-function c), the variables c_1, c_2, c_3 are passed as extra arguments to c_0, along with the "result" t_2 of the computation. Thus, the caller of f can put values into the arguments c_1, c_2, c_3 that it will need after f returns; it need not put those values into a closure-record in memory, with the expense of **fetch** and **store** instructions.

This is the entire essence of our callee-save representation. What remains is our method for organizing the closure-conversion algorithm to make use of this convention to best advantage. The main advantage is enhanced closure-sharing; figure 5 shows our new representation of the functions f and h.

Now the value w—needed by h—is passed as an argument (in a register) to g, from there to j_0 (where it's called j_2), then to h, without ever being

```
fun h(x,d0,w',d1,d2,d3) =
   let t1 = x*w'
    in d0(t1,d1,d2,d3)
   end

fun f'(f'',arg,c0,c1,c2,c3) =
     let val g = select(arg,0)
         val y = select(arg,1)

         fun j0(a,j1,j2,j3) =
              let val z' = select(j1,3)
               in h(z',k0,j2,j1,a,j3)
              end
         and k0(b,k1,k2,k3) =
                let val t2 = k2+b
                    val c0' = select(k1,0)
                    val c1' = select(k1,1)
                    val c2' = select(k1,2)
                 in c0'(t2,c1',c2',k3)
                end

         val w = select(f'',2)
         val z = select(f'',1)
         val u = (c0,c1,c2,z)
         val g' = select(g,0)
      in g'(g,y,j0,u,w,c3)
     end
```

Figure 5: Using 3 callee-save registers

stored into a closure. The variable a is passed from j_0 to h and then to k_0 in a register, so that now k and j can share a closure u. Note that some of the variables passed as callee-save arguments are ordinary variables (like w and a), and some are closure-records (like u); the compiler has great flexibility in using one or more of the callee-save registers to pass closure-records if necessary.

What we have achieved is that the functions j and k now share a closure, for a cost of $4B + C$ instead of $7B + 2C$. The fact that a is free in k and bound in j is no longer a problem, since a is passed from j to k in a callee-save argument (k_2).

Actually we can do even better than this. Since z and w are also free variables of function f, continuations j and k can directly grab the closure f'' so that we only need build a closure with $c0$, $c1$ and $c2$. Thus our cost is only $3B + C$:

```
fun f'(f'',arg,c0,c1,c2,c3)
     let ......
         val u = (c0,c1,c2)
         val g'= select(g,0)
      in g'(g,y,j0,u,f'',c3)
     end
```

In general, the new representation for continuations will save time and space when one function makes two or more non-tail calls. In the CPS representation, the continuations for these calls will be nested. The callee-save convention allows the continuation functions to be un-nested and to share a closure. Since all continuation functions are nested in some other user functions, the new representation for continuations can take advantage of the closure of the enclosing user functions if they happen to have some free variables in common, thus decreasing the cost of closure record constructions.

6.2 Strange continuation variables

As careful readers may have noticed, our mechanism will not work in the presence of first class continuations such as those introduced by call/cc and ML exceptions. A continuation obtained with call/cc may be put into a record, registered as an exception handler, stored into some reference cell, or passed as a "non-continuation" argument of an escaping user function. Thus, it must be representable in just one word, not as k separate variables. We can package a k-variable continuation into a single variable by making a record on the heap, as long as we remember to unpackage it at the right time.

In the CPS code before closure conversion, we say a continuation variable is *well-behaved* if all of its occurrences appear at the following positions:

- the second argument of escaping user functions;
- any argument of known functions;
- in function position, like g in $g(y)$.

All continuation variables that are not *well-behaved* are called *strange continuation variables*. In the example in figure 3, all continuation variables are well behaved.

To make our new schemes work correctly, we'll eliminate all strange continuation variables by transforming them into well-behaved continuation variables. First by using classical dataflow algorithm we can easily identify all continuation variables. Then for every strange occurrence v as in

1. `val r = (...,v,...)` where v is put in a record;
2. `k(v)` where an escaping continuation function k is applied to v;
3. `f(v,k)` where an escaping user function f is applied to (v, k);
4. `sethdlr(v)` where `sethdlr` is a primitive operator that registers v as an exception handler;

we define a new function u as `fun u(x,c) = v(x)` and substitute u for v:

1. ```
   let fun u(x,c) = v(x)
   in val r = (...,u,...)
   ```
2. `let fun u(x,c) = v(x) in k(u)`
3. `let fun u(x,c) = v(x) in f(u,k)`
4. `let fun u(x,c) = v(x) in sethdlr(u)`

so v is now well-behaved and u is just treated as an escaping user function.

For every strange occurrence v in

A. `val v = select(r,i);`

B. `fun k(v) = ...` where k is an escaping continuation function;

C. `fun f(v,k) = ...` where f is an escaping user function;
   ```

D. `val v = gethdlr()` where `gethdlr` is a primitive operator that grabs the current exception handler;

we substitute u for v and redefine v as a well-behaved continuation function by `fun v(x) = u(x,0)`, i.e.,

A. `val u = select(r,i); fun v(x) = u(x,0)`

B. `fun k(u) = let fun v(x) = u(x,0) in`

C. `fun f(u,k) = let fun v(x) = u(x,0) in`

D. `val u = gethdlr(); fun v(x) = u(x,0)`

Since the 2nd argument of u will never be used, we simply supply a 0. So v is well-behaved at these occurrences.

We can use similar methods to make every known function have at most one continuation argument. Thus we obtain *well-formed CPS expressions* that satisfy the following conventions:

- All escaping user functions will have two arguments, the first one is the standard non-continuation argument and the second is always a well-behaved continuation variable.
- All escaping continuation functions have only one, non-continuation, argument.
- Each known function has an arbitrary number of arguments but at most one of them is a continuation variable.

The well-formed CPS expression will be fed into the closure-conversion phase.

Now that we know a continuation variable can only appear at certain places, we can simply use $n+1$ actual parameters to represent each one as long as this number is consistent throughout the whole program.

6.3 Closure conversion strategies

To call a function, the actual parameters of the call must be put in registers first. Free variables of a known function can all be treated as extra actual parameters, unless the number of free variables is larger than the number of registers available on the target machine, so that we have to spill some of them into a closure. We can do such transformations at compile time.

The behavior of an escaping user function is not known at compile time. Because it might be put into and extracted from records etc., we can only use one single variable (i.e. the closure) to represent both the code pointer of the function and all its free variables. For escaping continuation functions, their call sites are not all known but they must be *well-behaved* in the sense defined in the previous section.

We now add a fixed number of parameters to each continuation function definition. The continuation function closure is now represented by one code pointer plus n extra variables (n must be same throughout the program but could be arbitrary from 1 to the maximum number of registers available in the target machine). These n extra variables behave just like callee-save registers. We denote them as $r_1, r_2, \ldots, r_n$.

Now every escaping continuation function will have n+1 arguments: the "return value" and the callee-save registers. Every escaping user function will have n+3 arguments (the closure record, the original (source-language) argument, the continuation code-pointer ("return address"), and the callee-save registers). All escaping functions use same calling conventions, so a fixed set of $n + 1$ registers can thus correspond to those $n + 1$ parameters for continuation code-pointers and callee-save registers.

All of the free variables of a continuation function must be accessible from the callee-save registers; this is the only "context" available to the function. But there will usually be more free variables than callee-save registers—especially because the free variables usually include the n callee-save registers of the enclosing continuation! Thus, one or more of the callee-save parameters must be *closure records* containing several free variables each.

To implement the callee-save approach, we must have an algorithm to decide how to arrange the free variables: some go in closure records and some can go directly in the callee-save parameters. Our choice will affect the number of continuations that can share closures (i.e. the number of closures built) and the size of closure records.

Our algorithm is not too complicated. We traverse the tree of nested functions (user and continuation), considering each node (set of mutually-recursive functions) in top-down order.

Note that the definition of a continuation function may be in the same set of mutually-recursive functions as a user function, though this is rare (it results from transformations done in previous phases of the compiler). In this case we simply make an ordinary closure containing the all free variables of all the functions in set. This is used as the closure-argument to user functions and as one of the callee-save arguments to the continuation function.

The more typical case is that we have an independent function definition of a single continuation.

We look to see if this continuation can use callee-save registers to avoid the need for a new closure. Consider a continuation k with a set F of free variables. The definition of k is lexically nested inside the definition of some other function p, which has closure(s) $C_1, C_2, \ldots$. If there is some set S of the C_i that efficiently cover F, we do not need a new continuation closure. That is, if

$$|S| \quad + \quad |F - \bigcup_{i \in S} C_i| \quad \leq \quad n$$

This says that the number of closures in S, plus the number of variables in F not covered by S, is not more than the number of callee-save registers. So the callee-save registers passed to k will hold the closures of S plus any free variables not covered by S.

Suppose there is no such set S; then we must make a new closure C, and pass it to k in one of the callee-save registers. An "aggressive" strategy would be to put *all* the free variables of k into C. This maximizes the chance, lower down in the tree, that some set S' (containing C) will cover the free variables of some other continuation. But it may also make C larger than necessary.

We know that C can contain as few as $|F| - n + 1$ variables: this is because one of the callee-save registers must hold the closure C, while the others are available to hold free variables; those free variables that don't fit must go into C. But which variables should we put into C, and which into the remaining $n - 1$ callee-save registers?

For *leaf* continuations—those containing no other continuations nested inside them—our answer is motivated by the desire to minimize "shuffling:" Those free variables that are callee-save parameters of a parent continuation we leave in callee-save registers, and the remaining free variables go into the closure. Then when it's time to invoke the parent, fewer **move** instructions will be needed. Of course, one of the callee-save parameters must go into the closure to make room in callee-save registers for the closure-pointer itself.

For *non-leaf* continuations we want to maximize the chance that some internally-nested continuation j can make use of C. We could do a reverse dataflow analysis to determine which free variables of k are free in j; these should go into C. Or we could just use the following simple heuristic:

Suppose k is nested inside some (user or continuation) function p; p's continuation and callee-save arguments are $a_0, a_1, \ldots, a_n$. Then the a_i are among the free variables of k. Now, since j must eventually call k, or call some function that calls k, we know that the a_i must be free in j as well.

But the other free variables of k might not be free in j.

So we should prefer putting the a_i in the closure, and the other free variables in callee-save registers.

6.4 Effects on garbage collection

The closures constructed using the callee-save conventions are smaller than those using "old-style" closure-passing style. Not only are the closure records smaller, but they keep fewer data structures live. Therefore we might expect that a copying garbage collector—which does work only proportional to the amount of live data—will have much less work to do on callee-save programs.

Regardless of the size of the *live data*, the callee-save convention should also create *less garbage*. This is because the closure records—which tend to become garbage quickly—are both smaller and fewer in number.

7. Register targeting

One would think that increasing the number of callee-save registers—while holding the number of caller-save registers fixed—can only improve the performance of the generated code. But in our initial implementation this did not seem to be the case. The problem turned out to lie in the algorithm for assigning CPS variables to registers of the target machine.

After CPS-conversion, closure-conversion, and spill transformation [4, 3], no subexpression of the CPS representation of the program can have more than m free variables, where m is the number of registers (callee+caller save) of the target machine. To do register assignment, we can traverse the CPS expression top-down, choosing a register for each variable-binding in turn. At each binding, e.g.

`let val c = a+b in` *subexpression* `end`

the free variables of *subexpression*, except for c, have already been assigned to registers; since there are no more than $m-1$ of them, we can always find some register to use for c.

No matter which register we pick, we can't run out of registers later on. But our choice of register affects the number of **move** instructions required later on: at a function call inside the *subexpression*, if c is an actual parameter of the call, we may have to move it to the register required for the formal parameter.

We would like to avoid such **move**s wherever possible. Before implementing the callee-save transformation described in this paper, we had three useful heuristics [3]:

1. Suppose there is a *known* function $f(\vec{x})$. We can choose registers in which the x_i are to be passed. On at least one of the calls to f, therefore, we won't need any **move** instructions at all: we'll just choose the same registers in which the actual parameters are located. This technique was used in Kranz's ORBIT compiler [16]. We can't do this for escaping functions, which must have standardized calling conventions.

2. Suppose the function call $f(\vec{x})$ within the *subexpression* is an escaping function, or a known function for which we have already determined the assignment of formal parameters. Then we check whether c is present among the x_i. If so, we can put c directly into the register required for the formal parameter, so that we won't need to **move** it later—this is called *targeting*. We can't do this if that register is currently occupied by some other variable, of course.

3. To avoid the problem alluded to in the previous sentence, we perform *anti-targeting*: if we are choosing a register for a variable-binding d, and d is *not* an actual parameter of a subsequent function call, we *avoid* putting d in any of the parameter registers of f unless we have no choice.

This set of heuristics had proved very effective. But with callee-save registers we found that an improvement was necessary: What does the targeting heuristic do for c if it finds a call (inside *subexpression*) to a known function $f(\ldots, c, \ldots)$ whose formal parameter register assignment has not been chosen? Our heuristics had previously assumed that any register could profitably be chosen for c. But we now find it necessary to look within the definition of f, to see whether there is a call $g(\ldots, c, \ldots)$ to a function g whose formal parameters have already been chosen; this will help in targeting c. The **move** instruction we avoid is not at the call to f, but at the call to g! Of course, if g is a function whose formal parameter register assignment has not yet been chosen, we must recur. We'll only a few times (we call it targeting depth). As indicated in the following section, depth-4 targeting can achieve good results.

What does this have to do with callee-save? Well, a very common case is that f is a known function, and g is a escaping function. Then the callee-save arguments of f should really be targeted to the callee-save registers required by g. Without this more sophisticated targeting, the cost of register-register *moves* is very high, as shown in figures 6 and 7.

	sml0			smlt0		
	time	icount		time	icount	speedup
Life	19.64	147578		19.51	147437	0.67%
Yacc	5.87	78875		6.08	78389	–3.58%
Lexgen	16.87	125186		17.25	123160	–2.25%
Knuth-B	16.27	223043		15.17	219778	6.76%
Simple	50.46	811066		50.82	800577	–0.71%
VLIW	35.29	362294		35.63	361355	–0.96%
	sml2			**smlt2**		
	time	icount	speedup	time	icount	speedup
Life	19.13	148673	2.60%	19.12	148514	2.65%
Yacc	5.40	72235	8.01%	5.33	71588	9.20%
Lexgen	15.93	117875	5.57%	15.41	114714	8.65%
Knuth-B	15.66	234495	3.75%	15.65	225204	3.81%
Simple	45.79	769750	9.25%	46.84	768252	7.17%
VLIW	32.51	351137	7.88%	32.96	348986	6.60%

Figure 6: MIPS 3230—running time in seconds, instruction counts in 1000's, and speedup of running time relative to **sml0**.

8. Benchmarks

The callee-save register technique was implemented by about 700 lines of ML code. We also made some minor modifications on the runtime system for the new calling conventions for continuation functions.

We ran eighteen versions of our compiler (Standard ML of New Jersey) on six different benchmark programs, on a MIPS 3230 workstation and on a SparcStation 2. The compilers were:

sml0 The "old-style" closure-passing style with basic register targeting.

smlt0 The "old-style" closure-passing style but with depth-4 register targeting.

sml2–9 With 2–9 callee-save registers, but only with basic register targeting.

smlt2–9 With 2–9 callee-save registers, but with depth-4 register targeting.

The benchmark programs were:

Life The game of Life, written by Chris Reade and described in his book [19], running 50 generations of a glider gun.

	sml0		**smlt0**	
	time	time		speedup
Life	27.72		27.29	1.55%
Yacc	9.22		8.88	3.69%
Lexgen	20.72		20.78	−0.29%
Knuth-B	23.66		24.30	−2.70%
Simple	70.77		71.93	−1.64%
VLIW	51.88		44.92	13.4 %
	sml3		**smlt3**	
	time	speedup	time	speedup
Life	26.87	3.1%	26.55	4.2%
Yacc	7.31	20.1%	6.88	25.4%
Lexgen	18.23	13.0%	18.17	12.3%
Knuth-B	22.10	6.6%	21.96	7.2%
Simple	58.54	17.3%	58.35	17.6%
VLIW	42.44	18.2%	41.58	19.9%

Figure 7: SPARC—running time in seconds and speedup relative to **sml0**.

Yacc A LALR(1) parser generator, implemented by David R. Tarditi [21], processing the grammer of Standard ML.

Lexgen A lexical-analyzer generator, implemented by James S. Mattson and David R. Tarditi [7], processing the lexical description of Standard ML.

Knuth-B An implementation of the Knuth–Bendix completion algorithm, implemented by Gerard Huet, processing some axioms of geometry.

Simple A spherical fluid-dynamics program, developed as a "realistic" FORTRAN benchmark [11], translated into ID [12], and then translated into Standard ML by Lal George.

VLIW A Very-Long-Instruction-Word instruction scheduler written by John Danskin.

Measurements on the machine such as MIPS 3230 became extremely inaccurate because of cache effects. If we change the location of the compiled code and the layout of the data space, the running time of the program can vary by 5% to 20% even though the same number of instructions is executed [3]. So we also measured instruction counts on the MIPS.

We found that two callee-save registers worked best on the MIPS, and three on the SPARC. Figure 6 gives the total running time (including garbage collection time) and instruction counts (not including garbage collection) by the compilers **sml0, smlt0, sml2, smlt2** on the MIPS 3230 workstation. Figure 7 gives the total running time of six benchmarks by the compilers **sml0, smlt0, sml3, smlt3** on a SparcStation. The instruction counts given in Figure 6 don't reflect the running time very well mainly because of the different instruction mixes in each situation. The callee-save techniques eliminated many **store** and **load** instructions but introduced more **move** instructions (register shuffling) because we stored many free variables in registers instead of on the heap. More advanced register targeting techniques greatly decrease the number of **move** instructions in some cases such as **Knuth-B**. In Figures 8 and 9, we plot the running time changes versus the number of callee-save registers with depth-4 register targeting on the MIPS and the SPARC. Complete measurements are given in figures 10–14.

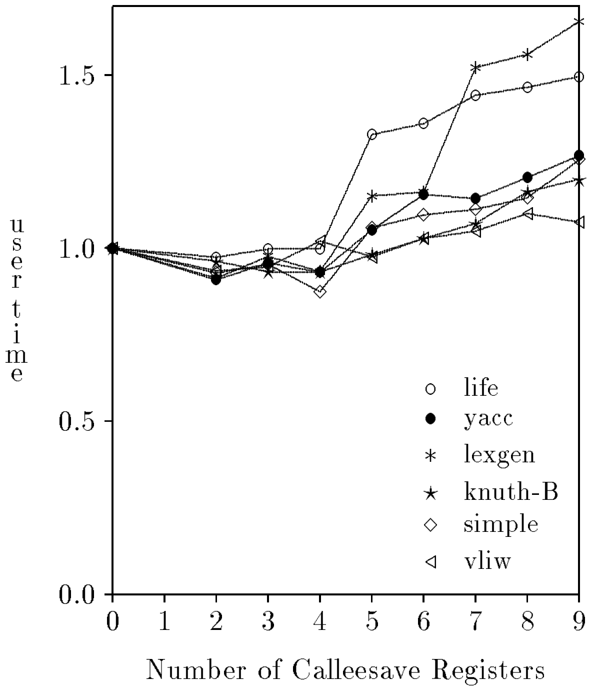


Figure 8: Execution time vs. number of callee-save registers on MIPS

If there are more than 5 callee-save registers, the performance suffers: this is probably because more and more data are spilled into closures since we have fewer registers available for non-callee-save purposes, it also could

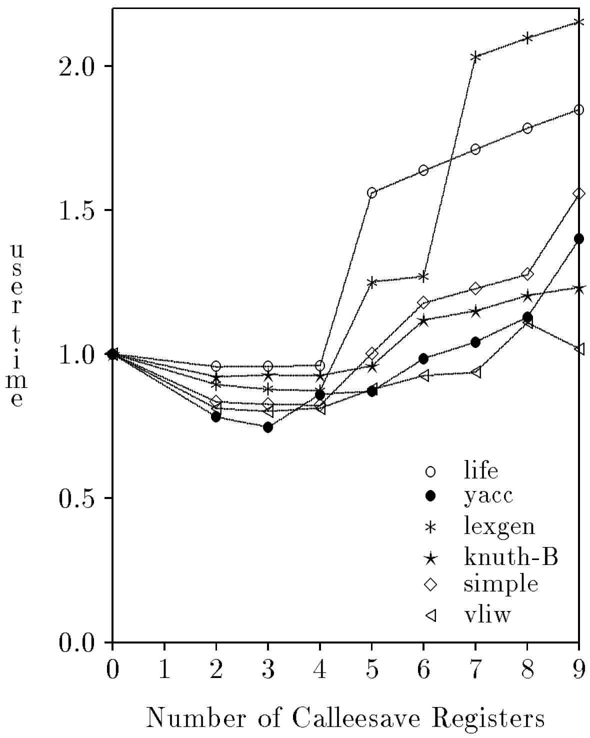


Figure 9: Execution time vs. number of callee-save registers on SPARC

be because we have more and more register shuffling as we use more callee-save registers. **Life** doesn't have significant speed-up because it has very few nested continuations.

In figures 10–14 we list the execution time, garbage collection time and total time of running **sml0, smlt0, sml2–9, smlt2–9** on all six benchmarks on both MIPS 3230 and SparcStation 2.

It turns out that both the compiled code and the garbage collection are sped up by using callee-save registers. In some cases, the garbage collection speeds up by a greater factor; in other cases, the compiled code.

Measurements of garbage collection time are very much dependent on heap size. For each benchmark, we used a heap of approximately 5 times the amount of live data when compiled by the **sml0** compiler; then the same heap size (regardless of the amount of live data) for the callee-save versions. We use an efficient two-generation collector[2].

9. Conclusions

Most continuation-based compilers[20, 17, 15] perform escape-analysis on closures to see which ones can be allocated on a stack. The idea is that the explicit deallocation (popping) of these closures is cheaper than garbage-collecting them. While we have argued that there is no inherent lower bound on the cost of garbage collection[1], clearly in real systems there is still some overhead (as figures 10–14 show). On the other hand, the use of a runtime stack has some serious disadvantages, adding to the complexity of the runtime system—especially when an efficient *call-with-current-continuation* is needed[13].

Our new *callee-save* technique will work well with or without stack allocation. The closures merged together by our method are a large proportion of the ones that would be stack allocated in the "traditional" method. Thus in the absence of a runtime stack, our method greatly reduces the amount of heap allocation; when a runtime stack is used, our method will not reduce heap allocation but will reduce the number of fetches and stores executed.

Our new callee-save version of the compiler generates code that runs up to 25% faster than the old version on the benchmark **Yacc**. Averaged over all the benchmarks our new version achieves a performance improvement over the already-efficient code generated by the old compiler—about 6.3 percent improvement on the MIPS and 14.4 percent on the SPARC.

10. Recent results and further work

John Reppy has recently improved the SPARC instruction scheduler used in our compiler, to avoid consecutive **store** instructions where possible. This gives a 5% overall improvement in run time. Since the instructions eliminated by callee-save continuations are mostly loads and stores, with our new scheduler one might expect that the improvements that callee-save gives on the SPARC will be slightly smaller than reported here.

We have recently noticed that closure sharing can lead to increased space usage. If continuations j and k share a callee-save closure record, and j has a free variable p not free in k, and if p is the root of a large (and otherwise dead) data structure, then p will cannot be garbage-collected until k is entered (and its closure becomes dead). This can seriously increase the memory usage of the program. However, it is still possible to share closures safely: in this example, p should be kept in a callee-save register instead of in the closure record. Thus, the solution to the problem is make sure that the earlier-dying values go into registers, and the later-dying values go into the closure record.

References

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

2. Appel, Andrew W. Simple generational garbage collection and fast allocation. *Software—Practice and Experience*, 19, 2 (1989) 171–83.

3. Appel, Andrew W. *Compiling with Continuations*. Cambridge University Press (1992).

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

5. Appel, Andrew W. and MacQueen, David B. A Standard ML compiler. In Kahn, Gilles, editor, *Functional Programming Languages and Computer Architecture (LNCS 274)*, Springer-Verlag, New York (1987) 301–24.

6. Appel, Andrew W., Ellis, John R., and Li, Kai. Real-time concurrent collection on stock multiprocessors. *SIGPLAN Notices (Proc. SIGPLAN '88 Conf. on Prog. Lang. Design and Implementation)*, 23, 7 (1988) 11–20.

7. Appel, Andrew W., Mattson, James S., and Tarditi, David R. A lexical analyzer generator for Standard ML. (December 1989). Distributed with Standard ML of New Jersey.

8. Cardelli, Luca. Compiling a functional language. In *1984 Symp. on LISP and Functional Programming*, ACM Press, New York (1984) 208–17.

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

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

11. Crowley, W. P., Hendrickson, C. P., and Rudy, T. E. *The SIMPLE Code*. Technical Report UCID 17715, Lawrence Livermore Laboratory, Livermore, CA (February 1978).

12. Ekanadham, K. and Arvind. *SIMPLE: An Exercise in Future Scientific Programming.* Technical Report Computation Structures Group Memo 273, MIT, Cambridge, MA (July 1987). Simultaneously published as IBM/T.J. Watson Research Center Research Report 12686, Yorktown Heights, NY.

13. Hieb, Robert, Dybvig, R. Kent, and Bruggeman, Carl. 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 (1990) 66–77.

14. Johnsson, Thomas. Lambda lifting: Transforming programs to recursive equations. In *The Second International Conference on Functional Programming Languag es and Computer Architecture*, Springer-Verlag, New York (September 1985) 190–203.

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

16. Kranz, David. *ORBIT: An optimizing compiler for Scheme.* PhD thesis, Yale University, New Haven, CT (1987).

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

18. Landin, P. J. The mechanical evaluation of expressions. *Computer J.*, 6, 4 (1964) 308–20.

19. Reade, Chris. *Elements of Functional Programming.* Addison-Wesley, Reading, MA (1989).

20. Steele, Guy L. *Rabbit: a compiler for Scheme.* Technical Report AI-TR-474, MIT, Cambridge, MA (1978).

21. Tarditi, David R. and Appel, Andrew W. ML-Yacc, version 2.0. (April 1990). Distributed with Standard ML of New Jersey.

	Life		Yacc	
	nongc+gc=total	icount	nongc+gc=total	icount
sml0	19.36+0.28=19.64	147578	5.31+0.56=5.87	78875
smlt0	19.22+0.29=19.51	147437	5.47+0.6 =6.08	78389
sml2	18.87+0.26=19.13	148673	4.86+0.54=5.4	72235
smlt2	18.8 +0.32=19.12	148514	4.8 +0.52=5.33	71588
sml3	18.97+0.29=19.27	148673	4.97+0.58=5.56	72861
smlt3	19.3 +0.28=19.58	148533	5.05+0.56=5.62	72092
sml4	18.9 +0.33=19.23	148801	5.08+0.59=5.68	75875
smlt4	19.33+0.25=19.59	148635	4.9 +0.55=5.46	74943
sml5	25.03+0.73=25.77	148530	5.46+0.54=6.0	82549
smlt5	25.33+0.75=26.08	148335	5.6 +0.58=6.18	81543
sml6	26.2 +0.7 =26.91	148198	5.75+0.56=6.31	86662
smlt6	26.03+0.68=26.71	148028	5.59+1.19=6.78	84681
sml7	27.0 +0.72=27.73	148478	6.4 +0.56=6.97	89887
smlt7	27.56+0.76=28.33	148292	6.16+0.54=6.71	87229
sml8	28.4 +0.43=28.83	147758	6.55+0.61=7.15	92420
smlt8	28.02+0.76=28.78	147702	6.44+0.63=7.07	90012
sml9	28.52+0.73=29.26	147799	7.27+0.63=7.91	101775
smlt9	28.61+0.76=29.38	147756	6.82+0.62=7.44	99890

Figure 10: MIPS3230—time in seconds and instruction counts in 1000's.

	Lexgen		Knuth-B	
	nongc+gc=total	icount	nongc+gc=total	icount
sml0	15.74+1.13=16.87	125186	12.98+3.28=16.27	223043
smlt0	15.73+1.51=17.25	123160	13.33+1.83=15.17	219778
sml2	15.34+0.59=15.93	117875	13.64+2.02=15.66	234495
smlt2	14.81+0.6 =15.41	114714	13.26+2.39=15.65	225204
sml3	15.32+0.58=15.9	117917	13.4 +2.16=15.57	237422
smlt3	15.46+1.01=16.47	116620	12.98+2.18=15.16	226933
sml4	15.25+0.57=15.83	120443	13.21+2.1 =15.32	239620
smlt4	15.13+0.59=15.72	118734	12.98+2.15=15.14	229534
sml5	20.11+1.73=21.84	132479	13.88+2.36=16.23	250352
smlt5	18.21+1.2 =19.41	128488	13.47+2.49=15.96	235782
sml6	19.13+0.67=19.8	138908	16.12+1.65=17.77	263367
smlt6	18.98+0.61=19.6	140314	14.64+2.08=16.72	251447
sml7	25.28+1.33=26.62	144016	16.71+2.13=18.84	264230
smlt7	24.93+0.73=25.67	141455	15.67+1.75=17.43	259299
sml8	32.6 +0.73=33.33	146169	16.39+2.85=19.23	282814
smlt8	24.93+1.39=26.32	145552	16.4 +2.52=18.93	275686
sml9	26.49+0.78=27.27	156092	17.4 +1.74=19.14	292532
smlt9	25.64+2.28=27.92	153010	16.88+2.61=19.5	281437

Figure 11: MIPS3230—time in seconds and instruction counts in 1000's.

	Simple		VLIW	
	nongc+gc=total	icount	nongc+gc=total	icount
sml0	42.57+7.88=50.46	81106	31.66+3.63=35.29	362294
smlt0	42.09+8.72=50.82	80057	32.01+3.61=35.63	361355
sml2	38.82+6.97=45.79	76975	29.27+3.24=32.51	351137
smlt2	38.92+7.92=46.84	76825	29.42+3.54=32.96	348986
sml3	39.91+8.75=48.66	77917	30.16+2.75=32.91	355794
smlt3	39.14+8.97=48.11	76663	30.26+3.11=33.38	353334
sml4	39.66+5.91=45.57	77754	32.29+3.83=36.12	365372
smlt4	38.99+5.1 =44.09	77168	32.62+3.35=35.98	362814
sml5	44.56+8.65=53.21	86950	31.45+3.97=35.41	380265
smlt5	44.68+8.72=53.41	86170	31.69+2.72=34.41	376616
sml6	48.77+6.1 =54.88	95397	32.98+2.9 =35.88	392013
smlt6	48.96+6.36=55.32	94677	32.75+3.51=36.27	386805
sml7	49.5 +5.65=55.16	98327	33.92+2.95=36.87	397194
smlt7	49.94+6.17=56.11	97404	33.8 +3.21=37.01	391090
sml8	51.21+7.33=58.54	100661	36.27+3.19=39.46	434974
smlt8	51.14+6.61=57.76	99375	35.74+3.07=38.81	432709
sml9	56.85+6.68=63.54	109882	34.34+3.28=37.62	408539
smlt9	56.68+6.72=63.4	108761	34.58+3.36=37.94	401969

Figure 12: MIPS3230—time in seconds and instruction counts in 1000's.

	Life		Yacc		Lexgen	
	nongc+gc	total	nongc+gc	total	nongc+gc	total
sml0	27.56+0.16	27.72	7.57+1.64	9.22	19.59+1.12	20.72
smlt0	27.16+0.13	27.29	7.6 +1.28	8.88	19.72+1.06	20.78
sml2	26.43+0.14	26.58	6.66+0.33	7.0	17.72+1.04	18.76
smlt2	26.34+0.17	26.51	6.84+0.37	7.21	17.47+1.06	18.53
sml3	26.71+0.16	26.87	6.56+0.74	7.31	17.55+0.68	18.23
smlt3	26.39+0.16	26.55	6.59+0.28	6.88	17.51+0.66	18.17
sml4	26.39+0.16	26.55	6.52+0.28	6.81	17.33+0.31	17.64
smlt4	26.39+0.2	26.59	6.45+1.48	7.93	17.34+0.72	18.06
sml5	42.99+0.22	43.22	7.7 +0.73	8.43	23.99+1.19	25.18
smlt5	42.99+0.24	43.23	7.72+0.3	8.03	24.82+1.07	25.9
sml6	45.0 +0.49	45.5	8.7 +1.15	9.86	25.87+0.78	26.66
smlt6	44.85+0.5	45.35	8.76+0.31	9.07	25.59+0.72	26.31
sml7	46.94+0.52	47.46	9.2 +0.31	9.52	41.01+1.22	42.23
smlt7	46.88+0.52	47.41	9.23+0.38	9.61	40.91+1.2	42.11
sml8	48.81+0.51	49.33	10.0+1.15	11.2	41.67+1.24	42.91
smlt8	48.97+0.48	49.46	9.6 +0.8	10.4	41.82+1.64	43.46
sml9	50.97+0.54	51.52	11.7+0.32	12.0	43.7 +1.23	44.93
smlt9	50.75+0.51	51.26	12.0+0.87	12.9	43.37+1.24	44.61

Figure 13: SPARC—running time in seconds.

	Knuth-B		Simple		VLIW	
	nongc+gc	total	nongc+gc	total	nongc+gc	total
sml0	22.46+1.2	23.66	64.44+6.33	70.77	49.9 +1.97	51.88
smlt0	22.36+1.94	24.3	65.64+6.29	71.93	43.72+1.2	44.92
sml2	20.43+1.72	22.15	54.76+5.19	59.95	41.0 +2.28	43.29
smlt2	20.16+1.63	21.79	54.21+4.79	59.00	40.05+2.02	42.07
sml3	20.79+1.31	22.1	54.33+4.21	58.54	40.6 +1.83	42.44
smlt3	20.37+1.59	21.96	54.11+4.24	58.35	39.89+1.69	41.58
sml4	20.48+1.69	22.17	52.91+3.75	56.66	41.01+1.86	42.87
smlt4	20.22+1.66	21.88	54.12+4.05	58.17	40.36+1.76	42.13
sml5	21.57+1.62	23.2	66.92+5.52	72.44	43.48+2.63	46.11
smlt5	21.17+1.51	22.68	66.72+4.19	70.91	43.73+1.77	45.51
sml6	24.72+2.07	26.8	79.08+3.63	82.71	46.3 +2.54	48.84
smlt6	24.44+2.02	26.46	79.76+3.55	83.31	46.02+1.97	47.99
sml7	25.47+2.02	27.49	82.68+4.26	86.94	45.79+1.79	47.58
smlt7	25.19+2.01	27.2	82.57+4.28	86.85	46.91+1.69	48.61
sml8	26.7 +1.71	28.41	86.56+4.25	90.81	60.1 +2.0	62.11
smlt8	26.49+1.97	28.46	86.15+4.22	90.3	55.84+1.68	57.52
sml9	27.37+1.88	29.26	104.4+3.79	108.2	51.72+2.22	53.94
smlt9	27.14+2.01	29.15	105.7+4.42	110.1	50.59+2.25	52.84

Figure 14: SPARC—running time in seconds.