Lambda, the Ultimate Label
or
A Simple Optimizing Compiler for Scheme

William D Clinger*
Lightship Software
OACIS
Tri-Step
University of Oregon

Lars Thomas Hansen
1626 Arthur Place
Eugene OR 97402
lth@cs.uoregon.edu

Abstract

Optimizing compilers for higher-order languages need not be terribly complex. The problems created by non-local, non-global variables can be eliminated by allocating all such variables in the heap. Lambda lifting makes this practical by eliminating all non-local variables except for those that would have to be allocated in the heap anyway. The eliminated non-local variables become local variables that can be allocated in registers. Since calls to known procedures are just gotos that pass arguments, lifted lambda expressions are just assembly language labels that have been augmented by a list of symbolic names for the registers that are live at that label.

1 Introduction

Twobit is a compiler for Scheme in the tradition of Rabbit, Orbit, and Gambit [23,16,9]. Unlike these previous compilers, which advanced the state of the art in generating efficient code, the main design goals for Twobit were simplicity, portability, and reasonably fast compilation, while generating code that is good enough for use in fairly high-performance systems comparable to Chez Scheme, Standard ML of New Jersey, and commercial implementations of Common Lisp. These goals have been met. The fundamental idea on which Twobit is built is that lambda expressions can be viewed as assembly language labels, and the formal parameters of a lambda expression can be viewed as an invariant that asserts the contents of general registers that are live at that label.

None of the optimizations used in Twobit are new, but their combined effectiveness has not been reported previously, nor has the use of single assignment analysis for first order closure analysis. As explained in Section 14, the flow equation used for lambda lifting in Twobit has some advantages over the similar equations in [1], especially for a simple compiler.

Twobit is biased toward RISC architectures. The primary goal of optimization in the front end is to reduce the problem of generating good code to a matter of register allocation and targeting. Currently the code generator is a simple, conventional generator that allocates registers as a stack and performs no register targeting apart from choosing an optimal order of evaluation for the operands of a procedure call (*parallel assignment optimization*). Twobit is designed to reward more sophisticated register targeting, and to make register allocation easy to express by adjusting the formal parameter lists of lambda expressions. Its design achieves some of the benefits of an intermediate form based on continuation-passing style (CPS) without actually requiring a conversion to CPS [23,1,18].

Simplicity is achieved by a front end whose two passes culminate in wholesale *lambda lifting*[1] [3,1]. This lambda lifting is made possible by a cascade of simpler optimizations, of which the most important is a first order closure analysis known as *single assignment analysis*. The output of the front end is an intermediate form in which lambda expressions are marked to indicate whether they correspond to assembly language labels (which represent both register allocation and control), to register allocation (`let`), or to closure allocation.

Portability is achieved by using a small, stylistically rigid subset of IEEE/ANSI Scheme [15] as the intermediate form, in which quoted data in command positions (where the value is ignored) and stylistic variations convey additional information to the code generator. This intermediate form can be compiled correctly by any Scheme compiler, but is designed as input to a code generator that will use the encoded information.

The current code generator generates assembly code for a hypothetical MacScheme machine similar to that described in [4], but register-based instead of stack-based. The MacScheme machine instructions have a semantics that was designed for effective peephole optimization. Table-driven optimizing assemblers currently generate byte code for an interpreter or machine code for the SPARC. Twobit has been used to construct an implementation of Scheme[2] known as Larceny, which has been used for research into the effect of programming style on the performance of Scheme programs [11].

*Current affiliation none of the below.

[1] Lambda lifting is called closure-conversion in [1]

[2] Larceny is a nearly complete implementation of IEEE/ANSI Scheme It is incomplete mainly because of some bugs in the bignum division routine

LISP 94 - 6/94 Orlando, Florida USA
© 1994 ACM 0-89791-643-3/94/0006..$3.50

```
Grammar for output of passes 1, 2, and 3:

L  -->  (lambda (I_1 ...)
          (begin D ...)
          (quote (R F <decls> <doc>)
          E)
     |  (lambda (I_1 ... . I_rest)
          (begin D ...)
          (quote (R F <decls> <doc>))
          E)
D  -->  (define I L)
E  -->  (quote K)              ; constants
     |  (begin I)              ; variable references
     |  L                      ; lambda expressions
     |  (E0 E1 ...)            ; calls
     |  (set! I E)             ; assignments
     |  (if E0 E1 E2)          ; conditionals
     |  (begin E0 E1 E2 ...)   ; sequential expressions
I  -->  <identifier>

R  -->  ((I <references> <assignments> <calls>) ...)
F  -->  (I ...)
```

Figure 1: The intermediate form.

2 Overview of Twobit

Twobit currently operates as a four-pass compiler. An optional fifth pass is planned but not yet implemented. The planned passes are

1. Standardization of syntax.
2. Optimization.
3. Representation inference (not yet implemented).
4. Code generation.
5. Assembly.

These passes will be discussed in order, using a definition of `reverse` as the main example:

```
(define reverse
  (lambda (x)
    (define (loop x y)
      (if (null? x)
          y
          (loop (cdr x) (cons (car x) y))))
    (loop x '())))
```

In Scheme, any of the standard procedures can be redefined, which in combination with separate compilation means that a compiler cannot generate inline code for calls to +, car, et cetera. Twobit, like most Scheme compilers, provides a compiler switch (`integrate-usual-procedures`) through which the programmer can promise not to redefine a subset of the standard procedures. The examples in this paper assume that switch is true, which is the default.

3 Pass 1: Standardization of syntax

Pass 1 expands macros, eliminates internal definitions, checks syntax, and gives a unique name to each local variable (*alpha conversion*). In addition, Pass 1 creates for each local variable a table R containing all references, assignments, and procedure calls to that variable.

```
((lambda ()
   (begin
    (set! reverse
          (lambda (.x_2)
            ((lambda (.loop_3)
               (begin
                (set! .loop_3
                      (lambda (.x_5 .y_5)
                        (if (null? .x_5)
                            .y_5
                            (.loop_3 (cdr .x_5)
                                     (cons (car .x_5)
                                           .y_5)))))
                ((lambda () (.loop_3 .x_2 '()))))
             '#!unspecified)))
    'reverse)))
```

Figure 2: The output of Pass 1.

The output of Pass 1 is expressed using the rigidly stylized subset of Scheme shown in Figure 1. Note especially that a variable `x` is represented by the equivalent `(begin x)`, which gives to variable references a list structure that can be shared and side effected. The following invariants hold for the output of Pass 1:

- There are no internal definitions.
- No identifier containing an upper case letter is bound anywhere.[3]
- No identifier is bound in more than one place.
- Each R contains one entry for every identifier bound in the formal parameter list and the internal definition list that precede the R. Each entry contains a list of pointers to all references to the identifier, a list of pointers to all assignments to the identifier, and a list of pointers to all calls to the identifier.
- Except for constants, the expression does not share structure with the original input or itself, except that the references and assignments in R are guaranteed to share structure with the expression. Thus the expression may be side effected, and side effects to references or assignments obtained through R are guaranteed to change the references or assignments pointed to by R.
- F is garbage.

The intermediate form is acyclic, and its printed form is genuine Scheme code, but it is large because of the shared structure and is hard to read because of all the clutter. Pass 1 converts the definition of `reverse` into an intermediate form equivalent to the code shown in Figure 2, which was produced by a `make-readable` procedure.

The `#!unspecified` notation in Figure 2, which stands for a canonical unspecified value, was produced by a particular implementation of the `letrec` macro. This notation is not a standard part of Scheme and is not treated specially by Twobit. Also, Scheme does not permit periods to begin identifiers, but some such illegal prefix is advisable to

[3] Scheme is not case-sensitive. This paper assumes an implementation that standardizes variable names to lower case, which allows the compiler to use upper case names for its own purposes. The handling of case is actually a parameter to the compiler

prevent the renamed local variables from shadowing global variables. The particular prefix used by Twobit is a parameter of the compiler.

Pass 1 was designed as an extension of an efficient algorithm for hygienic macro expansion [6,7], but the current implementation still uses non-hygienic macros.

4 Pass 2: Optimization

Most optimization occurs in Pass 2, whose structure is well explained by the Pass 2 procedure that optimizes lambda expressions:

```
(define (simplify-lambda exp notepad)
  (notepad-lambda-add! notepad exp)
  (single-assignment-analysis exp)
  (assignment-elimination exp)
  (let ((defs (lambda.defs exp))
        (body (lambda.body exp))
        (newnotepad (make-notepad exp)))
    (for-each (lambda (def)
                (simplify-lambda (def.rhs def)
                                 newnotepad))
              defs)
    (lambda.body-set! exp
                      (simplify body newnotepad))
    (lambda.F-set! exp
                   (notepad-free-variables newnotepad))
    (lambda-lifting exp (notepad.parent notepad))
    exp))
```

The notepad is a data structure used to pass inherited attributes and to record synthesized attributes, such as the set of lambda expressions that are nested within the enclosing lambda expression.

Single assignment analysis identifies any formal parameters that are assigned exactly once, at the head of the lambda body, to the result of a lambda expression, and are called as often as they are referenced. Such parameters are actually the names of local procedures whose call points are all visible to the compiler. There is no need to create a closure for such procedures, since the environment in which they are declared will be accessible from the environment in effect at each place where they are called. This fact is recorded by transforming

```
(lambda (... I ...)
  (begin D ...)
  (quote (... (I <references>
                 ((set! I L))
                 <calls>)
              ...))
  (begin (set! I L) E1 ...))
```

into

```
(lambda (... IGNORED ...)
  (begin (define I L) D ...)
  (quote (... (I <references> () <calls>)
              ...))
  (begin E1 ...))
```

in which the single assignment has become an internal definition. For example, the assignment to `loop` in Figure 2 becomes an internal definition in the output from Pass 2 (Figure 3).

Since the single assignment probably resulted from the elimination of an internal definition during Pass 1, it may seem that nothing has been accomplished. The key is that an internal definition in the output of Pass 2 records information gained by a simple closure analysis, whereas an internal definition in the original code may define a variable whose value is not a procedure, a variable whose value is changed by assignments, or a procedure that must be represented as a closure because it escapes.

A variable defined by an internal definition in the output of Pass 2 is a *known local procedure*. In Twobit, single assignment analysis also checks to make sure every call to a known local procedure passes the correct number of arguments. If the known local procedure has a rest parameter, the rest parameter is replaced by an ordinary parameter and the excess arguments in each call to that known procedure are replaced by appropriate calls to `CONS` (in upper case because it refers to the standard `cons` procedure, not to the value of the `cons` variable when the call is evaluated).

After single assignment analysis, *assignment elimination* finds any remaining local variables that appear on the left hand side of an assignment. If any such variables `I1 ...` are found, then the body of the lambda expression being optimized is replaced by a `let` of the form

```
((lambda (V1 ...)  ...)  (MAKE-CELL I1) ...),
```

in which all references to such variables are replaced by calls to `CELL-REF`, and all assignments by calls to `CELL-SET!`. The `MAKE-CELL` procedure allocates heap storage for the variable, which is essentially replaced by a pointer to that storage. Assignment elimination makes all local variables immutable as in Standard ML, so they can be copied freely, which greatly simplifies the lambda lifting that will conclude Pass 2.

Following assignment elimination, each known local procedure is optimized recursively, as is the body of the lambda expression being optimized. Several simple but important local source transformations come into play here. Twobit currently performs eight different transformations on `if` expressions, three different transformations on `let` expressions (procedure calls whose operator is an explicit `lambda` expression), flattening of nested `begin` expressions, and removal of constants and variable references that are evaluated only for effect. These local source transformations are essentially the same as in Rabbit [23].

One transformation on `let` expressions is especially important. When single assignment analysis detects a known local procedure, it replaces the parameter that was assigned once by the special parameter `IGNORED`, and assignment elimination replaces any parameters that are neither referenced nor assigned by the special parameter `IGNORED`. One local source transformation therefore replaces

```
((lambda (IGNORED I2 ...) <body>) E1 E2 ...)
```

by

```
(begin E1 ((lambda (I2 ...) <body>) E2 ...))
```

except the transformation applies to any `IGNORED` parameter, not just the first. If `E1` is a constant, which it is likely to be if the `IGNORED` parameter resulted from single assignment analysis, then the transformed `let` will just become another `let` that binds fewer variables. Eventually there may not be any variables left, in which case the original `let` will be transformed into its body after any internal definitions are lifted to an enclosing lambda expression.

After the known local procedures and the body of a lambda expression have been optimized, the free variables that were encountered are recorded as the free variables of the lambda expression.

Finally it is possible to lift the internal definitions of known local procedures to the enclosing lambda expression. For example, the internal definition of .loop_3 in

```
((lambda ()
   (begin (set! reverse
                (lambda (.x_2)
                  (define .loop_3
                    (lambda (.x_5 .y_5)
                      (if (null? .x_5)
                          .y_5
                          (.loop_3 (cdr .x_5)
                                   (cons (car .x_5)
                                         .y_5)))))
                  (.loop_3 .x_2 '())))
          'reverse)))
```

is lifted to the outer lambda expression as in Figure 3.

Although the reverse example makes lambda lifting appear easy, it is actually the most complex optimization performed by Twobit. The problem is that one of the known local procedures may be lifted past a parameter that occurs free in the local procedure being lifted. If so, then that parameter must be added as an extra parameter to the local procedure, and all calls to it must be changed to pass that extra parameter. Consider

```
((lambda ()
   (begin (set! foo
                (lambda (n x y)
                  (define (f n)
                    (if (zero? n) x (g (- n 1))))
                  (define (g n)
                    (if (zero? n) y (f (- n 1))))
                  (f n)))
          'foo)))
```

If f were lifted to the outer lambda expression, it would have to take g and x as extra parameters. That means g would have to supply g and x as extra arguments when it calls f:

```
((lambda ()
   (define (f g x n)
     (if (zero? n) x (g (- n 1))))
   (begin (set! foo
                (lambda (n x y)
                  (define (g n)
                    (if (zero? n)
                        y
                        (f g x (- n 1))))
                  (f g x n)))
          'foo)))
```

Notice that g now escapes its scope, negating the earlier closure analysis. Worse yet, if g were lifted to the outer lambda expression, it would have to take an extra parameter y. That means f would have to supply y as an extra argument when it calls g. But f is now outside the scope of y!

The solution is to lift each group of known local procedures as a unit, after performing a flow analysis to determine precisely which parameters must be added to each procedure. Let V be the set of parameters for the lambda expression at whose head a group of known local procedures is defined, let $f_1, \ldots$ be the names of those procedures, and let F_i be the variables that occur free in the body of f_i. The set of variables A_i that need to added as parameters to f_i is defined by the recursive flow equations

$$A_i = (F_i \cap V) \cup (\bigcup \{A_j \mid f_i \text{ calls } f_j\})$$

These equations are solved in Twobit by supplying suitable arguments to the procedure shown in Figure 4.

If $A_i = V$, then Twobit adds the extra parameters to the beginning of the parameter list, sorted so that, for a call to f_i from the body of the lambda expression that declares f_i, these parameters will already reside in the correct registers. This often eliminates register shuffling altogether. If $A_i \neq V$, then the order should be chosen to minimize register shuffling, but Twobit currently does nothing particularly intelligent in this case. It may at times be desirable to add dummy parameters so that, for example, the real arguments are passed in registers r6, r1, and r3 instead of r1, r2, and r3.

It is always possible to lift all known local procedures to the outermost lambda expression. This is not always desirable, however, because the formal parameters that must be added represent extra registers that must be saved across a non-tail-recursive call. There are many factors that the compiler could take into account when deciding whether to lift a group of known local procedures. Currently Twobit always lifts unless the body of the lambda expression contains a lambda expression for which a closure must be created anyway. This has worked surprisingly well in practice.

The output of Pass 2 is expressed using the same subset of Scheme used to express the output of Pass 1 (Figure 1). The following invariants hold for the output of Pass 2:

- There are no assignments except to global variables.
- If I is declared by an internal definition, then the right hand side of the internal definition is a lambda expression and I is referenced only in the procedure position of a call.
- For each lambda expression, the associated F is a list of all the identifiers that occur free in the body of that lambda expression, and possibly a few extra identifiers that were once free but have been removed by optimization.
- Variables named IGNORED are neither referenced nor assigned.

```
((lambda ()
   (define .loop_3
     (lambda (.x_5 .y_5)
       (if (null? .x_5)
           .y_5
           (.loop_3 (cdr .x_5)
                    (cons (car .x_5) .y_5)))))
   (begin (set! reverse
                (lambda (.x_2) (.loop_3 .x_2 '())))
          'reverse)))
```

Figure 3: The output of Pass 2.

```
; Given a vector of starting approximations,
; a vector of functions that compute a next
; approximation as a function of the vector of
; approximations, and an equality predicate,
; returns a vector of fixed points.

(define (compute-fixedpoint v functions equiv?)
  (define (loop i flag)
    (if (negative? i)
        (if flag
            (loop (- (vector-length v) 1) #f)
            v)
        (let ((next_i ((vector-ref functions i) v)))
          (if (equiv? next_i (vector-ref v i))
              (loop (- i 1) flag)
              (begin (vector-set! v i next_i)
                     (loop (- i 1) #t))))))
  (loop (- (vector-length v) 1) #f))
```

Figure 4: A general routine for solving flow equations.

5 Pass 3: Representation inference

This optional pass, which is not implemented, will infer representations using a flow algorithm based on conjunctive types. The main goal of this pass is to resolve dynamically overloaded operators statically and to eliminate run-time type checking without compromising safety.

6 Pass 4: Code generation

This pass is being rewritten to reuse continuation frames across more than one procedure call, as described in Section 9. The current generator keeps all variables in registers or in the heap, using a stack cache only for continuations needed by non-tail-recursive calls and as a place to spill registers. Registers are spilled on the fly by allocating a new stack frame for the spilled registers. This was a major design error, for reasons explained in Section 8.

The remainder of this section describes the current code generator.

The code generator generates assembly code for a hypothetical MacScheme machine. The MacScheme machine architecture uses a single representation, a *closure*, for both procedures and heap-allocated environments. There are four circumstances in which code is generated to create a closure, of which only the first corresponds to creating a procedure rather than a heap-allocated environment. A closure is created:

1. for any lambda expression that is neither the right hand side of an internal definition nor the operator of a procedure call (`let`);
2. upon entry to any lambda expression that has internal definitions, unless the body is also a lambda expression (this is explained below);
3. upon entry to any lambda expression that takes more than a certain number of parameters (because this means there are few registers available, which makes spilling likely, and the spill code generated by the current generator is quite poor);
4. when there are not enough registers available to allocate the variables declared by a `let`.

When the code generator is rewritten, the third and fourth reasons to create a closure will be eliminated, and a stack frame will be used instead. The second reason will remain, as there are several reasons why stack frames generally cannot be used to hold variables that occur free in known local procedures:

- The known local procedure may be referred to within some lambda expression that escapes.
- The use of a stack frame to hold the free variables of a recursive procedure interferes with tail recursion, as explained in [12]. See Section 10.
- The `call-with-current-continuation` may be implemented in such a way that stack frames can be moved dynamically, which makes non-local addressing problematic.

The current code generator performs only a few optimizations, nearly all of which are trivial. *Let optimization* simply allocates registers for the variables declared by a `let`. *Procedure integration* generates inline code for calls to a set of integrable procedures defined by a table in the compiler. To some integrable procedures there may correspond an immediate form of a MacScheme machine instruction that implements the procedure; this immediate form will be generated if the corresponding operand is a suitable constant. The code generator also implements a very special case of *common subexpression elimination*: if an operand is a variable that already resides in a register, then no code will be generated for that operand.

The most interesting optimization performed by the code generator is *parallel assignment optimization*, which attempts to find an order of evaluation for the operands of a procedure call that allows all of the operands to be evaluated directly into the registers that will be used to pass the arguments. This optimization is performed by constructing a dependency graph in which register i depends on register j if and only if register j contains a free variable of the ith operand. A topological sort of this dependency graph yields an optimal order of evaluation. Twobit currently gives up and resorts to wholesale register shuffling if the graph contains a cycle, but it would be better to remove one of the registers involved in the cycle and to continue the sort.

Parallel assignment optimization works very nicely with the register allocation performed by the front end during lambda lifting. Taken together, lambda lifting and parallel assignment constitute an effective strategy for global register allocation in Scheme-like languages. Despite the crudeness of Twobit's current heuristics for register allocation and parallel assignment, parameters that are added during lambda lifting usually reside in the target registers when the lifted procedure is called. The design of Twobit, in particular its view of lambda expressions as assembly language labels augmented by register contents, will reward substantial efforts to improve its heuristics for register allocation.

7 Pass 5: Assembly

The output from the code generator is assembly code for the MacScheme machine, a general register plus accumulator architecture that was specifically designed for peephole optimization and translation into real machine languages. A table-driven optimizing assembler currently generates byte

```
000000  b5000100  lambda  ^000028,0,(#t)
000004  5b0004    setglbl reverse
000007  5c0004    const   reverse
00000a  10        return
00000b  1f        nop
00000c  4ea8ffff  .entry            ; loop
000010  c1        reg     1         ; x
000011  5e19      op1     null?
000013  bf0002    branchf ^000018   ; branch if false
000016  c2        reg     2         ; y
000017  10        return
000018  c1        reg     1         ; x
000019  5e1b      op1     car
00001b  62a8      op2     cons,2
00001d  e2        setreg  2         ; next y
00001e  c1        reg     1         ; x
00001f  5e1c      op1     cdr
000021  e1        setreg  1         ; next x
000022  baffeb    branch  ^000010   ; tail-recursive
000028
000028  4ea8ffff  .entry            ; reverse
00002c  5001      args=   1
00002e  a2        const   ()
00002f  e2        setreg  2
000030  b7000c01  jump    1,$000c   ; jump to loop
```

Figure 5: MacScheme machine code for **reverse**.

code for an interpreter, and a similar table-driven assembler in Larceny generates SPARC machine code.[4]

Figure 5 shows the disassembled byte code for **reverse**. The first two instructions create a closure and store it in the global variable **reverse**. The entry point for **reverse** is at address 000028. The closure for the currently executing procedure is always kept in general register 0, so register 0 will contain the closure for **reverse** when control passes to this entry point. Since closures double as environments, general register 0 is also the environment register through which any non-local and non-global variables would be accessed, though in this case there are none.

The argument to **reverse** is passed in general register 1, and an argument count is passed in the accumulator, which is called the **result** register. After checking to make sure exactly one argument was passed, the **const** instruction at address **2e** loads **result** with the empty list, which is then moved to general register 2. The next instruction jumps to the entry point for the known local procedure **loop**. As indicated by its first operand, the **jump** instruction follows one link of the environment chain before taking the jump, which places the closure for **loop** in general register 0.

At address 10 the **loop** procedure tests its first argument **x**, branching to address 18 if **x** is not the empty list. If **x** is the empty list, however, then the **result** register is loaded with the value of **y** from general register 2 and control returns to the current continuation.

The tail-recursive call to **loop** begins at address 18. Parallel assignment optimization has determined that it is better to evaluate the second argument first, so the **car** of **x** is computed into the **result** register by instructions at 18-19. The instruction at **1b** conses that result onto the value of **y** from general register 2, and the **setreg** instruction at **1d** moves the resulting pair into general register 2. Instructions **1e-21** then evaluate the **cdr** of **x** into general register 1. The call is completed by branching to the first instruction in the body of the **loop** procedure.

For the SPARC, the Larceny assembler performs peephole optimization and fills branch delay slots. Most branch delay slots are filled quite easily by moving the target of the branch into the delay slot, adding 4 to the target address of the branch, and changing the branch into an annulled branch (indicated by the **.a** suffix) so that the delay slot will be executed *only if the branch is taken.* Figure 6 shows the disassembled SPARC code for the known local procedure **loop**, with comments showing the MacScheme machine instructions from which the SPARC code was generated. For example,

```
80  subicc   $r.reg1 10 $r.g0  ; reg 1; op1 null?
```

SUBtracts the Immediate operand 10 from the value in the MacScheme machine's general register 1 and discards the result by sending it to the SPARC's global register 0, which is hardwired to zero, because the purpose of this instruction is to set the Condition Codes. Since 10 is Larceny's representation for the empty list, this single SPARC instruction corresponds to the first two MacScheme machine instructions in the body of **loop**.

At first glance it appears that instructions are out of order in

```
92  ldi      $r.stkp 0 $r.o7          ; return
96  jmpli    $r.o7 8 $r.g0
100 orr      $r.reg2 $r.g0 $r.result  ; reg 2
```

The first instruction fetches the return address from the stack cache, and the second instruction jumps to that address plus 8. The third instruction ORs the value in the MacScheme machine's general Register 2 with zero, effectively copying it to the MacScheme machine's **result** register. Larceny's assembler has moved this instruction into the delay slot for the **jmpli** instruction that returns from **loop**. These examples illustrate the effectiveness of peephole optimization on MacScheme machine assembly code.

The **$r.millicode** notation stands for a SPARC register that points to a jump table of low-level routines for storage allocation, exception handling, and so forth.

Optimizing Scheme compilers are traditionally tested by examining the code they generate for a tight, empty loop. Larceny keeps a countdown timer in a register, checking for interrupts when the counter reaches zero, which adds one cycle to most loops. For example, the loop expressed by

```
(let ((x 1)
      (y 2)
      (tight-loop 3)
      (z 4))
  (set! tight-loop
        (lambda (a b c) (tight-loop a b c)))
  (tight-loop x y z))
```

compiles into

```
100     subicc   $r.timer 1 $r.timer
104     bne.a    104
108     subicc   $r.timer 1 $r.timer
```

Here the instruction that decrements the countdown timer has been copied into the branch delay slot, resulting in the tightest loop possible for the SPARC architecture: two cycles.

[4] There is no good reason why there are two separate assemblers. The person who wrote the second assembler has regretted it since.

```
80      subicc   $r.reg1 10 $r.g0              ; reg 1; op1 null?
84      bne.a    104                          ; branchf 104
88      nop
92      ldi      $r.stkp 0 $r.o7              ; return
96      jmpli    $r.o7 8 $r.g0
100     orr      $r.reg2 $r.g0 $r.result      ; reg 2
104     andi     $r.reg1 7 $r.tmp0            ; BEGIN: reg 1; op1 car
108     xoricc   $r.tmp0 1 $r.g0              ;   (tag check)
112     be.a     132                          ;   (skip and load if ok)
116     ldi      $r.reg1 -1 $r.result         ;
120     jmpli    $r.millicode 80 $r.o7        ;   (exception: not a pair)
124     addi     $r.o7 -24 $r.o7              ;
128     ldi      $r.reg1 -1 $r.result         ; END: reg 1; op1 car
132     orr      $r.result $r.g0 $r.argreg2   ; BEGIN: op2 cons,2
136     jmpli    $r.millicode 16 $r.o7        ;   (allocate 8 bytes)
140     ori      $r.g0 8 $r.result
144     sti      $r.result 0 $r.argreg2       ;   (store CAR)
148     sti      $r.result 4 $r.reg2          ;   (store CDR)
152     addi     $r.result 1 $r.result        ; END: op2 cons,2  (tag PTR)
156     orr      $r.result $r.g0 $r.reg2      ; setreg 2
160     andi     $r.reg1 7 $r.tmp0            ; BEGIN: reg1; op1 cdr; setreg 1
164     xoricc   $r.tmp0 1 $r.g0
168     be.a     188
172     ldi      $r.reg1 3 $r.reg1
176     jmpli    $r.millicode 80 $r.o7
180     addi     $r.o7 -24 $r.o7
184     ldi      $r.reg1 3 $r.reg1            ; END:  reg1; op1 cdr; setreg 1
188     subicc   $r.timer 1 $r.timer          ; BEGIN: branch 80
192     bne.a    84
196     subicc   $r.reg1 10 $r.g0
200     jmpli    $r.millicode 168 $r.o7       ;   (timer expired)
204     addi     $r.o7 -128 $r.o7             ; END: branch 80
```

Figure 6: SPARC machine code for `loop` (local to `reverse`).

8 First class continuations

Scheme's `call-with-current-continuation` procedure implies that most continuations have potentially unlimited extent, which means they cannot simply be stack-allocated. Larceny uses the incremental stack/heap strategy to deal with first class continuations. This strategy has precisely the same performance as conventional stack allocation when `call-with-current-continuation` is not used [5]. We say that the incremental stack/heap strategy has *zero overhead*. Zero overhead is achieved by allocating continuation frames in a stack cache, and by keeping a dummy continuation frame at the bottom of the stack cache. When a procedure returns into the dummy frame, which can happen only after the stack cache has been cleared,[5] the return address in the dummy frame gives control to a stack cache underflow handler that copies one continuation frame from the heap into the stack cache, and then returns through the copied frame.

For this to work, every continuation frame must correspond to some procedure call. For example, this zero-overhead strategy would be upset by a code generator that allocates a separate stack frame to hold registers that must be spilled. Twobit's original code generator did this.

If a spill frame is allocated that does not correspond to a procedure call, and this allocation is followed by the allocation of a continuation frame for a non-tail-recursive call, and the stack cache is flushed during that call, then the underflow handler will copy the continuation frame into the stack cache following the call but will not copy the spill frame. When the continuation frame is popped following the return, the stack cache will become empty and the spill frame will be unaddressable.

Several solutions are available. Instructions could be generated to check for an empty stack cache whenever registers need to be restored from a spill frame, which is rare. Instructions could be generated to check for an empty stack cache whenever frames are popped, but in Larceny this solution would cost four extra cycles on every non-tail-recursive procedure call. The best solution is to avoid the creation of separate spill frames, which is the solution we will adopt when we rewrite the code generator in order to reuse continuation frames for more than one non-tail-recursive call.

[5] The stack cache is cleared when a continuation becomes a first class object. In Larceny the stack cache can also be cleared by a stack cache overflow, task switch, or garbage collection These other causes reflect design choices, however, as it is not really necessary for any of them to clear the stack cache.

9 Reuse of continuation frames

Reuse of continuation frames is important for procedures like doubly recursive `fib` that contain more than one non-tail-recursive call. Reuse is made more difficult by the fact that we do not want to allocate a continuation frame on entry to `fib` because it is important to keep the base cases for a recursion as simple as possible. The continuation frame should be allocated only if needed. In other words, the frame should be allocated as late as possible to make sure it is actually needed, but it should also be allocated as early as possible to ensure that it is shared by all the non-tail-

```
E  -->  (quote K)              E.ntemps = -1.

E  -->  (begin I)              E.ntemps = -1.

E  -->  L                      E.ntemps = -1.

E  -->  (E0 E1 ...)            E0.save = E1.save = ... = false;
                               E1.nregs = 0;
                               E2.nregs = 1;
                               ...
                               E0.nregs = # of args;
                               if E.tail
                                 then E.ntemps = max(E0.ntemps, E1.ntemps, ...)
                                 else E.ntemps = max(0,
                                                     E0.ntemps, E1.ntemps, ...) + E.nregs.

E  -->  (set! I E0)            E0.save = E.save; E0.nregs = E.nregs;
                               E.ntemps = E0.ntemps.

E  -->  (if E0 E1 E2)          E0.save = false;
                               E0.nregs = E1.nregs = E2.nregs = E.nregs;
                               if E0.ntemps >= 0
                                 then E1.save = E2.save = false
                                 else E1.save = E2.save = E.save;
                               E.ntemps = max(E0.ntemps, E1.ntemps, E2.ntemps).

E  -->  (begin E0 E1 E2 ...)   E0.save = E1.save = E2.save = ... = false;
                               E0.nregs = E1.nregs = E2.nregs = ... = E.nregs;
                               E.ntemps = max(E0.ntemps, E1.ntemps, E2.ntemps, ...).
```

Figure 7: Attribute grammar for reusing continuation frames.

recursive calls. True optimization of this tradeoff is statically undecidable.

The algorithm planned for Twobit maintains the following invariant: There are never two or more continuation frames allocated at the same time by a single procedure activation. Code will be generated as though a continuation frame has been allocated, and pop instructions will be emitted before tail-recursive calls and returns, relying on the assembler to suppress the pop if the value of its symbolic operand is negative, indicating that no frame was ever actually allocated.

Two new attributes will be computed by the code generator. The synthesized attribute **ntemps** is the number of temporary locations that might be needed by an expression; its value will be -1 for an expression that not only requires no temporaries, but also requires no continuation frames. The inherited attribute **save** is a boolean that indicates whether an expression is responsible for allocating a frame itself if one is necessary for that expression. The existing inherited attribute **nregs** indicates how many registers are live.

These attributes are computed according to the attribute grammar in Figure 7, which is simplified by omitting other attributes, by ignoring the issue of register spills, and by assuming that no procedures are integrable and that operands are evaluated from left to right. A continuation frame will be allocated when:

- E is `(E0 E1 ...)` and `E.ntemps` ≥ 0 and `E.save` = `true`.
- E is `(if E0 E1 E2)` and `E0.ntemps` ≥ 0 and `E.save` = `true`.
- E is `(begin E0 E1 E2 ...)` and `E.ntemps` ≥ 0 and `E.save` = `true`.

This algorithm is suboptimal for examples like

```
(+ x (if debugging (read) 1))
```

but it has the desirable property that a continuation frame will never be allocated except when a frame is potentially necessary, and no more than one frame can ever be allocated as part of the activation of a procedure. Furthermore this algorithm does the right thing for `fib`, `map`, and most other common examples.

Although this is a fairly complex algorithm, it appears that its net effect will be to simplify the code generator, because registers will be spilled to stack frames in a simple, uniform way.

10 Tail recursion

Stack allocation of non-local variables makes proper tail recursion harder to implement, because a tail-recursive call to a local procedure cannot safely deallocate a stack frame when that frame contains non-local variables that are needed by the procedure being called. Consequently a return from a procedure may involve deallocating stack frames that were allocated by neither the returning procedure nor by the procedure to which it is returning [12].

Lambda lifting as in Twobit eliminates this problem by eliminating non-local variables, except for those variables that must be allocated in the heap anyway as part of a closure. Figure 8 shows a troublesome example from [12], and Figure 9 shows that same example after lambda lifting.

The fact that Twobit currently doesn't use the stack for local variables is irrelevant. Twobit *will* allocate some local variables on the stack when the code generator is rewritten.

```
(define accumulate
  (lambda (binary-op initial items)
    (if (null? items)
        initial
        (letrec
          ((loop
            (lambda (value items)
              (if (null? items)
                  value
                  (loop (binary-op      ; on the stack?
                                value
                                (car items))
                                (cdr items))))))

          ; This next call should be properly
          ; tail recursive.

          (loop (car items) (cdr items))))))
```

Figure 8: Interference between stack allocation and tail recursion.

```
((lambda ()
   (define .loop_3
     (lambda (.binary-op_2 .value_5 .items_5)
       (if (null? .items_5)
           .value_5
           (.loop_3 .binary-op_2
                    (.binary-op_2 .value_5
                                  (car .items_5))
                    (cdr .items_5)))))
   (begin
    (set! accumulate
          (lambda (.binary-op_2 .initial_2 .items_2)
            (if (null? .items_2)
                .initial_2

                ; No problem with tail recursion here.

                (.loop_3 .binary-op_2
                         (car .items_2)
                         (cdr .items_2)))))
    'accumulate)))
```

Figure 9: The accumulate example after lambda lifting.

The real reason Twobit is able to avoid this problem with tail recursion is that all *non-local* variables live in the heap.

11 Performance of Larceny

Larceny was written by Hansen under Clinger's direction to study the effect of programming style on performance, especially the performance of garbage collection [11]. Figure 10 compares the performance of Larceny with that of three other systems on several small benchmarks described in the next section. By comparing system performance we hope to place the issue of compiler optimization in context. We believe the costs of procedure calls, storage allocation, and garbage collection are more important than compiler optimization for this sort of language. Our goal is not to show that Twobit is the best compiler—we know, for example, that Standard ML of New Jersey has a substantially more sophisticated compiler—but to show that simple optimizing

	Larceny	Chez Scheme	Allegro CL	SML/NJ
fib30	6.9	7.0	6.3	8.3
cpstak	.47	.43	—	
reverse				
10000 x 100	2.3	3.0	2.8	1.3
100 x 10000	3.9	4.5	5.8	2.2
native reverse				
10000 x 100		1.8	—	1.3
100 x 10000		2.7	6.2	2.3
reverse!				
10000 x 100	1.4	5.1	1.4	
100 x 10000	1.5	5.1	1.4	
native reverse!				
10000 x 100		1.6	1.7	
100 x 10000		1.6	1.8	
append-1				
10000 x 100	4.6	4.3	21.4	4.0
100 x 10000	30.2	12.2	20.0	6.6
append-2				
10000 x 100	3.5	6.5	4.0	
100 x 10000	5.1	8.6	6.9	
native append				
10000 x 100		3.6	4.0	2.5
100 x 10000		10.7	7.5	4.5
idiv2	.33	.57	.53	
rdiv2	.54	.65	2.18	
boyer	5.6	4.0	5.2	
perm8	3.7	3.2	4.0	
mergesort	9.3	15.4	21.6	
quick-1	6.0	6.8	8.7	7.2
recursive-nfa	2.2	2.5	6.2	4.1
sieve-4	1.4	1.5	8.3	2.3
puzzle	3.2	6.5	9.2	

Figure 10: Times in seconds for several small benchmarks.

compilers such as Twobit can be good enough to serve as the foundation for an efficient implementation.

Of the benchmarks reported in Figure 10, the best tests of compiler optimization occur toward the bottom. The times reported in Figure 10 represent CPU time in seconds, including garbage collection, for the median (Larceny) or second fastest (all others) of at least three runs. More runs were used when anomalous or variable timings were observed. All measurements were performed on owyhee, a Sun-4/40 (SPARCstation IPC). No declarations were used except for compiler switches, which were set to generate the fastest reasonably safe code (the only kind Larceny can generate at present). Generic arithmetic was used except in Standard ML.

Larceny can be configured with any of three distinct garbage collectors. The times shown in Figure 10 are for Larceny version 0.15 with the generational collector, which is most similar to the collectors used in the other systems. The default compiler switches were used.

Chez Scheme 4.1 was timed with `(optimize-level 2)`.

Allegro Common Lisp 4.1 [SPARC; R1] (3/17/93 20:38) was timed with

```
(proclaim '(optimize (speed 3) (safety 1)
                     (space 0) (debug 0)))
```

Standard ML of New Jersey 0.93 was timed with the default compiler switches, of which it is said "There is little point in fiddling with these flags to improve the performance of the optimizer" [2].

12 Benchmarks

Source code for most of these benchmarks can be found in [11] or [10]. To allow for the failure of tail recursion in Common Lisp, the Common Lisp versions of append-2, reverse, reverse!, perm8, and sieve-4 were rewritten to use do and/or prog instead of tail recursion. The Standard ML benchmarks use pattern matching where possible.

Fib30 is a doubly recursive computation of the 30th Fibonacci number. Cpstak is a translation of the tak benchmark [10] into continuation-passing style as a test of closure allocation. The reverse benchmarks allocate one million pairs by calling the reverse procedure (as in this paper) either 10000 times (10000×100) or 100 times (100×10000). The native reverse benchmark calls the predefined reverse instead, which is presumably written in a style that is close to optimal for a particular system. The reverse! benchmark is similar but uses a destructive reversal that allocates no storage. Append-1 uses a deeply recursive version of append, while append-2 uses an iterative version that performs destructive operations on newly allocated pairs. Idiv2 and Rdiv2 are widely reported benchmarks of simple iterative and recursive list processing [10]. Boyer is another garbage collection benchmark based on a theorem prover [10]. With the exception of fib30 and reverse!, these benchmarks have more to do with storage allocation and garbage collection than with the compiler.

Perm8 uses Zaks's algorithm to create a list of the 40320 permutations of a list of eight integers in grey code order, allocating 149912 pairs to represent that list. The perm8 benchmark creates no garbage whatsoever. Mergesort sorts the list created by perm8 using a destructive merge sort that allocates no storage. Quick-1 is an in-place quicksort of a vector of 30000 randomly chosen nonnegative integers. Recursive-nfa is 1000 calls to a recursive implementation of a string pattern matcher. Sieve-4 uses thunks as generators to compute the first 800 primes. Puzzle is written in the style of the Pascal benchmark from which it was translated [10].

The last six benchmarks offer the greatest opportunities for compiler optimization. As may be inferred from our analysis in the next section, however, there is no such thing as a pure compiler benchmark for this class of programming languages.

13 Analysis of benchmark results

Larceny, Chez Scheme, and Standard ML of New Jersey are properly tail recursive and eschew the SPARC's register windows, while Allegro Common Lisp uses the register windows for non-tail-recursive calls but does not handle tail recursion properly in general.[6] It follows that Allegro Common Lisp has an advantage on benchmarks that use shallow recursion (fib30) but is at a disadvantage on benchmarks that use deep recursion (append-1, rdiv2) or tail recursion (cpstak, mergesort).

Larceny and Chez Scheme both allocate continuation frames in a stack cache [5,13], but Standard ML of New Jersey allocates all continuation frames in the heap, relying on sophisticated garbage collection to make this strategy practical [1]. This becomes evident on the fib30 benchmark.

[6] The anti-correlation between tail recursion and register windows is not accidental. The SPARC's standard calling conventions use the register windows but render a fully general implementation of proper tail recursion impractical.

To simplify its three interchangeable garbage collectors, Larceny flushes the stack cache on every garbage collection. The cost of this becomes apparent on the append-1 benchmark with a recursion depth of 10000. Allegro Common Lisp performs poorly on this benchmark even with a recursion depth of 100 because owyhee's processor has only 20 register windows. Standard ML of New Jersey does well on this benchmark because it does *not* use a stack cache.

We observed excessive garbage collection when Allegro Common Lisp's native reverse procedure was called repeatedly on lists of length 100. Its performance was almost uniformly better on lists of length 10000, contrary to expectation.

By default, Twobit compiles self-recursive calls to global procedures that are written in a certain style as if the procedure will never be redefined. This benchmark-mode may give Larceny an unfair advantage on rdiv2 and a small advantage on boyer. The other benchmarks are not affected by this compiler switch.

Chez Scheme and Allegro Common Lisp use a "function cell" for faster calls to global procedures, which improves performance on the boyer benchmark.

Although the mergesort benchmark itself allocates no storage, its side effects may require a generational garbage collector to record cross-generational pointers. All four of the benchmarked systems use such a collector. Larceny's superior performance on mergesort may be accounted for in part by its large ephemeral area, which at one megabyte is large enough to accommodate most of the storage for the permutations being sorted. Side effects into the ephemeral area do not have to be recorded. We observed times ranging from 7.9 to 10.4 seconds on this benchmark, apparently depending on how much of the storage had been promoted to an older generation. This phenomenon may benefit both Larceny and Standard ML of New Jersey on the quick-1 and puzzle benchmarks. We do not know the size of the ephemeral area used by Chez Scheme and Allegro Common Lisp.

Although the mergesort benchmark neither allocates nor deallocates storage, it reveals one of the true costs of garbage collection. In C a side effect is cheap. With generational collectors, side effects are expensive [11].

On the sieve-4 benchmark, Larceny generates better code than Standard ML of New Jersey because of a subtle difference in the flow equations used for lambda lifting. This is discussed in the next section.

14 Related work

Twobit is heir to a distinguished family of compilers that began with Guy Steele's Rabbit [23], which introduced alpha conversion, local source transformations, and several other optimizations used in Twobit. Steele is also responsible for the view that a procedure call is simply a goto that passes arguments [22,24,20,21], from which it follows that a lambda expression is simply a label that can accept arguments.

Lambda lifting, as a practical compilation technique, was introduced by Augustsson for Lazy ML, who noted an analogy with combinator abstraction but gave few details [3,14, 25]. Lambda lifting is more radical than hoisting in Standard ML of New Jersey (SML/NJ) or lambda drifting in Liar, neither of which will lift a lambda expression outside the scope of one of its free variables [1,19]. Lambda

lifting corresponds instead to the closure-conversion[7] algorithm of SML/NJ [1]. The flow equation used for lambda lifting in Twobit is more local than the equations used in SML/NJ, since it involves only those variables whose scope is being left by the lifted procedures. This makes it easy to end the lifting short of the outermost lambda expression, which allows greater sharing of environment structure than in SML/NJ. Twobit's local, incremental approach to lambda lifting seems simpler, may be faster, and can at times generate better code than the global, all-or-nothing approach taken by SML/NJ. The flow equation for lambda lifting in Twobit was developed independently of [1].

The semantics of assignment in Scheme makes assignment elimination, introduced by the Orbit compiler [16], a prerequisite to lambda lifting. Orbit was probably the first Scheme compiler to view the evaluation of arguments as a parallel assignment to registers, and to optimize their order.

Single assignment analysis has long been used in MacScheme and possibly in Orbit, but has not been described heretofore. The MacScheme machine architecture used by Twobit is descended from Scheme 311, and from an analysis of what is wrong with the architecture used in MacScheme versions 2 through 4 [4,17]. The incremental stack/heap strategy for continuations also comes from MacScheme [5].

Conversion to continuation-passing style (CPS) has been advocated by several authors of optimizing compilers [23, 16,1]. Despite the documented advantages of CPS, Clinger prefers direct style as a more readable intermediate form. Premature conversion to CPS makes register allocation, targeting, and parallel assignment optimization more difficult. Since these were the focus of Twobit, it seemed better to use direct style. Nothing in the design of Twobit precludes a conversion to CPS by the code generator, but this would add an extra pass.

The machine-independent part of Twobit contains less than 5000 lines of code, which is less than one tenth the size of some compilers. The machine-specific assembler, disassembler, and linker for the SPARC also come to less than 5000 lines. Twobit is much less ambitious than Python [18] or the compiler for Standard ML of New Jersey [1], but achieves its limited ambitions: simplicity, portability, and efficient generation of efficient code.

15 Conclusion

Twobit is simple because all non-local and assigned variables are allocated on the heap. Twobit generates reasonably efficient code because lambda lifting makes most variables local, and because a heuristic ordering of the parameters for each lambda expression, combined with parallel assignment optimization, often yields a good interprocedural allocation of registers. Twobit is portable because it is written in Scheme, its intermediate language is Scheme, and it is easy to change the target of the MacScheme machine assembler. Compared to other optimizing compilers, Twobit is fairly fast because it consists of a small fixed number of passes, the first of which gathers information needed for optimization while performing macro expansion.

The most important idea in Twobit is that lambda lifting converts a known local procedure into an assembly language label that has been augmented by a description of the registers that are live at that label.

[7] To add to the confusion, [26] uses "closure conversion" to refer to a source code transformation that replaces a procedure by some representation of the procedure. This transformation is orthogonal to lambda lifting.

16 Acknowledgements

This research was supported in part by a grant from OACIS and from Lightship Software.

Our special thanks go to Gene Luks for reconstructing Zaks's algorithm.

References

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

[2] AT&T Bell Laboratories. Standard ML of New Jersey system modules (version 0.93). February 15, 1993.

[3] Lennart Augustsson. A compiler for Lazy ML. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming,* pages 218–227, August 1984.

[4] Clinger, W.D. The Scheme 311 compiler: an exercise in denotational semantics. *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming,* August 1984, pages 356–364.

[5] Clinger, W.D., Hartheimer, A.H., and Ost, E.M. Implementation strategies for continuations. *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming,* pages 124–131, July 1988.

[6] Clinger, W., and Rees, J. Macros that work. *Proceedings of the 1991 ACM Conference on Principles of Programming Languages,* pages 155–162, January 1991.

[7] Clinger, W., and Rees, J. [editors]. Revised[4] report on the algorithmic language Scheme. *Lisp Pointers* 4(3), July–September 1991, pages 1–55.

[8] R. Kent Dybvig. *The Scheme Programming Language.* Prentice-Hall, 1987.

[9] Marc Feeley and James Miller. A parallel virtual machine for efficient Scheme compilation. *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming,* June 1990.

[10] Richard P. Gabriel. *Performance and Evaluation of Lisp Systems.* MIT Press, 1985.

[11] Lars Thomas Hansen. *The Impact of Programming Style on the Performance of Scheme Programs.* M.S. thesis, University of Oregon, 1992.

[12] Chris Hanson. Efficient stack allocation for tail-recursive languages. *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming,* June 1990.

[13] Robert Hieb, R. Kent Dybvig, and Carl Bruggeman. Representing control in the presence of first-class continuations. *PLDI '90, SIGPLAN Notices,* 25(6), pages 66–77, June 1990.

[14] John Hughes. Super combinators: a new implementation method for applicative languages. *Proceedings of the 1992 Symposium on Lisp and Functional Programming,* pages 122–132.

[15] *IEEE Standard 1178-1990. IEEE Standard for the Scheme Programming Language.* IEEE, New York, 1991.

[16] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin, and Norman Adams. Orbit: an optimizing compiler for Scheme. *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, SIGPLAN Notices*, 21(7), pages 219–233, July 1986.

[17] Lightship Software, Inc. *MacScheme* Version 4 software and manual. Lightship Software, 1992.

[18] Robert A MacLachlan. The Python compiler for CMU Common Lisp. *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming, LISP Pointers*, 5(1), pages 235–246, June 1992.

[19] Guillermo Juan Rozas. Taming the Y operator. *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming, LISP Pointers*, 5(1), pages 226–234, June 1992.

[20] Guy Lewis Steele Jr. and Gerald Jay Sussman. Lambda, the ultimate imperative. MIT Artificial Intelligence Memo 353, March 1976.

[21] Guy Lewis Steele Jr. Lambda, the ultimate declarative. MIT Artificial Intelligence Memo 379, November 1976.

[22] Guy Lewis Steele Jr. Debunking the "expensive procedure call" myth, or procedure call implementations considered harmful, or lambda, the ultimate GOTO. *ACM Conference Proceedings*, pages 153–162. ACM, 1977.

[23] Guy Lewis Steele Jr. Rabbit: a compiler for Scheme. MIT Artificial Intelligence Laboratory Technical Report 474, May 1978.

[24] Guy Lewis Steele Jr. Compiler optimization based on viewing LAMBDA as RENAME + GOTO. In *AI: An MIT Perspective.* Patrick Henry Winston and Richard Henry Brown, editors. MIT Press, 1980.

[25] D.A. Turner. New implementation techniques for applicative languages. *Software—Practice and Experience*, 9, pages 31–49, 1979.

[26] Mitchell Wand and Paul Steckler. Selective and lightweight closure conversion. *Proceedings of the 1994 ACM Symposium on Principles of Programming Languages*, pages 435–445, January 1994.