ORBIT: An Optimizing Compiler for Scheme

David Kranz*, Richard Kelsey*, Jonathan Rees#
Paul Hudak*, James Philbin*, and Norman Adams+

*Yale University
Department of Computer Science
Box 2158 Yale Station
New Haven, CT 06520

#Mass. Institute of Technology
Artificial Intelligence Lab
545 Technology Square
Cambridge, MA 02139

+Tektronix, Inc.
Beaverton, OR 97077

1. Introduction

In this paper we describe an optimizing compiler for Scheme [3, 13] called *Orbit* that incorporates our experience with an earlier Scheme compiler called TC [10, 11], together with some ideas from Steele's Rabbit compiler [14]. The three main design goals have been correctness, generating very efficient compiled code, and portability.

In spirit, Orbit is similar to the Rabbit compiler in that it depends on a translation of source code into "continuation-passing style" (CPS), a convenient intermediate form that makes control-flow explicit. After CPS conversion, procedures take an extra argument called a *continuation* (another procedure) that represents the next logical execution point after execution of the procedure body. Thus procedures do not "return," but rather "continue into" the code represented by the continuation. This permits, for example, a general but simple way to optimize tail-recursions into loops.

Steele's seminal work on Rabbit demonstrated the general benefits of this approach to compiler design. However, his work was primarily research oriented, and Rabbit was essentially a prototype compiler (consider, for example, that it generated MACLISP code). TC, on the other hand, was one of the first *practical* compilers for a Scheme dialect, and much was learned through its design and construction.[2] Orbit now represents a culmination of that learning process, in which CPS conversion has been implemented thoroughly, extended in critical ways, and set in a framework of other important compiler innovations to yield a practical compiler that generates production-quality code competitive with the best compilers for Lisp as well as non-Lisp languages.[3] The new ideas in Orbit include:

1. An extension to CPS conversion called *assignment conversion* that facilitates treatment of assigned variables. (See section 3.)
2. Compact and efficient runtime data representations, the most important being those for lexical closures, which are represented in different ways depending on information gleaned from a static analysis of their use. (See sections 4 and 5.)
3. A register allocation strategy based on *trace scheduling* that optimizes register usage across forks and joins (and which, because of CPS conversion, includes procedure calls). (See section 6.)
4. An integral table-driven assembler that uses hints from the code generator to order blocks so as to minimize the number and size of jumps. (See section 7.)
5. A technique for defining the behavior of primitive operations in Scheme source modules rather than in the internal logic of the compiler. (See section 9.1.)
6. Flexibility in configuring the runtime support system for user programs. (See section 9.2.)

[1]This work was supported in part by NSF Grants DCR-8403304 and DCR-8451415, and a Faculty Development Award from IBM. In addition, some of the work was supported at DEC Western Research Laboratory in the summer of 1984.

[2]Both TC and the S-1 Common Lisp compiler described in [2] have as a common ancestor a Lisp compiler which Steele wrote during the summer of 1979. From this compiler TC was derived by making modifications for compilation of a different source language for different target machines, and adding the logic necessary for compiling lexical closures moderately well.

Despite these innovative aspects of Orbit, to treat them in isolation is to miss one other key feature in the overall design; namely, the way in which the pieces fit together. It was only through experience with the TC compiler that we were able to make the right choices for data representation, requirements for flow analysis, and strategies for code generation.

2. Preliminary Benchmarks

Since "the proof is in the pudding," we present here some benchmarks of Orbit compiled Scheme code, and compare them with benchmarks for several other languages and compilers. Benchmarks are usually placed at the end of papers, but by placing them here we hope to motivate the reader to actually reach the end of the paper.

The benchmark results for Orbit presented below are preliminary - many standard optimizations have yet to be included in the compiler. For example, the compiler currently does no sophisticated flow analysis, and thus it does no optimizations such as common sub-expression elimination, loop invariance, or loop induction. Nevertheless, on these benchmarks, Orbit currently produces code that is competitive with and in many cases superior to comparable compiler/machine combinations. More importantly, the benchmarks demonstrate that a language based on full lexical closures can be compiled efficiently.

Our benchmarks were chosen from Richard Gabriel's textbook on evaluating Lisp systems [7].[4] In tables 2-1 and 2-2 below the benchmark numbers for lisp systems other than Orbit come from that book. The Orbit benchmarks were run using open coded (i.e., non-atomic) storage allocation.[5]

Table 2-1 compares benchmarks of Orbit running on a Sun 3/160M with 4 megabytes of memory, with benchmarks of four other Lisps and machines: a Symbolics 3600 with an IFU and an unspecified amount of memory running Zetalisp, a Xerox Dorado with 2 megabytes of memory running Interlisp, and a Vax 8600 and Vax 780 with unspecified amounts of memory running DEC Common Lisp. There are too many variables in table 2-1 to draw firm conclusions about one implementation and machine versus another, but we believe that it does demonstrate that an optimizing compiler can produce code running on stock hardware that is competitive with code running on specialized hardware.

Program	Orbit (SUN III)	3600 +IFU	Dorado	8600 (Dec CL)	780 (Dec CL)
Tak	0.25	0.43	0.52	0.45	1.83
Takl	1.63	4.95	3.62	2.03	7.34
Boyer	15.84	9.40	17.08	12.18	46.79
Browse	40.28	21.43	52.50	38.69	118.51
Destructive	1.24	2.18	3.77	2.10	6.38
Deriv	3.62	3.79	15.70	4.27	13.76
Dderiv	4.92	3.89	17.70	6.58	19.00
IDiv2	0.24	1.51	3.43	1.65	5.00
RDiv2	0.36	2.50	4.08	2.52	9.84
Triangle	84.36	116.99	252.20	99.73	360.85
Fprint	2.18	2.60	2.93	1.08	3.94
Fread	2.62	4.60	1.57	2.34	7.24
Tprint	1.66	4.89	5.55	0.70	2.85

Table 2-1: Orbit vs. Other Lisp Engines

[4]We have not included all of the benchmarks in Gabriel's suite. Notably, STAK, CTAK, FFT and POLY were omitted because they test aspects of Orbit (dynamic binding, downward continuations, and floating point arithmetic) that are currently implemented in a preliminary and inefficient manner. Other benchmarks were omitted either because they seemed inconsequential (such as TAKR), because data for all the lisp systems was unavailable, or because of lack of time.

[5]Since Orbit is designed to support multi-processing, it can optionally allocate storage atomically, but it is slightly more expensive to do so.

Table 2-2 shows a comparison of Orbit with Franz Lisp and PSL. The entries are elapsed CPU time in seconds.[6] Both the Orbit and PSL benchmarks were run on an Apollo DN300 with 1.5 megabytes of memory. Franz was run on a SUN II with an unspecified amount of memory.

Program	Orbit DN300	PSL DN300	Franz Sun II
Tak	1.34	1.62	2.37
Takl	6.21	12.90	12.82
Boyer	63.16	46.92	37.94
Destructive	7.91	10.16	9.57
Dderiv	28.12	28.95	16.95

Table 2-2: Orbit vs. PSL vs. Franz

We have also run two preliminary benchmarks used at DEC Western Research Laboratory [9], PERM and TOWERS, and compared the results against those using C, Pascal, and Modula II. The Orbit benchmarks were run on a Vax/750, while the other benchmarks were run on a Vax/780. The Orbit numbers presented in table 2-3 are normalized to 780 equivalent numbers using the assumption that a 750 is equivalent to 60% of a 780. These results lead us to believe that modern lisps can indeed be compiled to produce code that is competitive with that produced by compilers for conventional algol-like languages. Further results of these benchmarks will be presented in a forthcoming research report.

Program	Orbit	Unix C	DEC C	DEC Pascal	DEC Modula II
Perm	1.26	2.6	2.5	2.5	2.0
Tower	1.65	2.6	2.7	2.6	1.9

Table 2-3: Orbit vs. Algol-like Languages

Overview of Paper

There are nine phases in the Orbit compiler:

1. Alpha conversion
2. CPS conversion
3. Assignment conversion
4. Optimization
5. Declaration generation
6. Closure strategy analysis
7. Closure representation analysis
8. Code generation
9. Assembly

The first five phases are discussed in the next section, and the remaining four are discussed in sections 4 through 7. In section 8 we present some detailed examples of compiled code.

3. Extended CPS Conversion

3.1. Alpha Conversion

We assume that the reader is familiar with alpha conversion, which is basically a renaming of bound variables. Variables with identical names and different scopes are renamed. This allows the optimization phase of the compiler to ignore the problem of variable name conflicts (i.e., aliasing). The conversion is described in detail in [14].

[6] CPU time on the Apollo includes page fault handler overhead, while CPU time on the Sun does not. Because of this anomaly benchmarks that are "cons intensive" (Boyer, Destructive, and Dderiv) are not comparable between the Apollo and the Sun. Further since the amount of memory on the Sun II system is not specified in Gabriel's book we have no idea whether Franz was subject to a similar number of page faults as PSL and Orbit.

3.2. CPS Conversion

The use of CPS conversion in Orbit differs from Rabbit in two important ways: First, *assignment conversion* eliminates the need for special (usually ad hoc) treatment of assignments (via `set!`) to local variables. Second, all high-level optimizations are done on CPS code, not on source code.

In the following description of the translation process, we assume that the reader is familiar with conventional Lisp notation; in particular, the use of quote and backquote, where, for example, `` `(a ,b c) `` is equivalent to `(list 'a b 'c)`. The CPS conversion algorithm takes as input two code fragments (i.e., syntactic objects) - an *expression* and its *continuation* - and returns a third code fragment representing the CPS code.

If the expression is an atom it is simply passed to the continuation:

```
    (convert <atom> <cont>) => `(,<cont> ,<atom>)
```

If the expression is an application then the procedure and all the arguments must be converted. The continuation is introduced as the first argument in the final call:

```
    (convert '(<proc> . <arguments>) <cont>) =>
      (convert <proc> `(lambda (p) ,(convert-arguments <arguments> `(,p ,<cont>))))

    (convert-arguments '() <final-call>) => <final-call>

    (convert-arguments '(<argument> . <rest>) <final-call>) =>
      (convert <argument>
         `(lambda (k) ,(convert-arguments <rest> (append <final-call> '(k)))))
```

Special forms have their own conversion methods. `lambda` expressions have a continuation variable added which is used in converting the body:

```
     (convert '(lambda (a b c) <body>) <cont>) =>
       `(,<cont> (lambda (k a b c) ,(convert <body> 'k)))
```

The first expression in a block is given a continuation containing the second expression.

```
     (convert '(begin <exp1> <exp2>) <cont>) =>
       (convert <exp1> `(lambda (v) ,(convert <exp2> <cont>)))
```

Conditional expressions are transformed into calls to a `test` primitive procedure which takes two continuations. This eliminates the need for a primitive conditional node type in the code tree, while preserving the applicative-order semantics of CPS code [13]. A new variable `j` is introduced to avoid copying the continuation.

```
     (convert '(if <exp1> <exp2 <exp3>) <cont>) =>
       (convert <exp1> `(lambda (v)
                          ((lambda (j)
                             (test (lambda () ,(convert <exp2> 'j))
                                   (lambda () ,(convert <exp3> 'j))
                                   v))
                           <cont>)))
```

Here is an example of CPS conversion (in which all of the trivial beta-substitutions have been done for clarity):

```
      (convert '(lambda (x y) (if (= x y)
                                  (write x)
                                  (begin (write x) (write y))))
               'k0)

    =>  (k0 (lambda (k1 x y) (= L1 x y)))
        L1 =  (lambda (v1) ((L2 k1)))
        L2 =    (lambda (k2) (test L3 L4 v1))
        L3 =      (lambda () (write k2 x))
        L4 =      (lambda () (write L5 x))
        L5 =        (lambda k3 (write k2 y))
```

To make the code easier to read, note that the individual lambda expressions are written and named separately. This linear representation reflects the sequential flow of control that the CPS conversion makes explicit. Care must be taken when reading this code, however, since the introduced names are intended to denote textual substitution; that is:

```
L1 =  (lambda (v1) ((L2 k1)))
L2 =    (lambda (k2) (test L3 L4 v1))
```

is the same as:

```
L1 =  (lambda (v1) (((lambda (k2) (test L3 L4 v1)) k1)))
```

Thus `v1` in the body of L2 is bound by L1.

3.3. Assignment Conversion

Assignment conversion is done after CPS conversion, and amounts to introducing *cells* to hold the values of assigned variables.[7] This conversion is best explained by showing it in Scheme source code rather than CPS code:

```
(lambda (x)  ... x ... (set! x value) ...)
  => (lambda (x)
       (let ((y (make-cell x)))
         ... (contents y) ... (set-contents! y value) ...))
```

where `(make-cell x)` returns a new cell containing the value `x`, `(contents cell)` returns the value in `cell`, and `(set-contents! cell val)` updates `cell` with the new value `val`. After assignment conversion, the values of *variables* can no longer be altered - all side-effects are to data structures. This greatly simplifies the code transformation (optimization) phase, because values may now be freely substituted for variables without having to first check to see whether they are assigned.

After alpha, CPS, and assignment conversion, there are only three kinds of nodes in intermediate code trees: LAMBDA-nodes, applications, and value nodes (variables and constants). This small number of node types allows Orbit to concentrate on generating high quality code for a small number of primitives. In particular, it concentrates attention on the representation and manipulation of *closures*, which are used pervasively in Scheme in a large number of ways. For example, they are used not only as continuations resulting from CPS conversion, but also as source level continuations obtained through `call-with-current-continuation`. Such "first-class continuations" allow the implementation of complex control structures such as coroutines that are normally found primitive in other languages [6, 8]. Closures are also used as *data structures*, in which the environment that is "closed over" essentially contains the elements of the data structure [1]. The importance of the closure in Scheme cannot be overemphasized, and thus the ability to generate good code for it is essential for a good Scheme compiler. Orbit's treatment of closures is discussed in detail in sections 4 and 5.

3.4. Program Optimization

Most of the optimizations in Orbit are accomplished via transformation, including partial evaluation, of the intermediate CPS code. The simplicity, regularity, and explicit nature of CPS code make this strategy superior to the alternatives of either transforming source code or performing similar optimizations at the object code level.

Partial evaluation is compile-time evaluation of certain expressions that are normally evaluated at run-time. Other than constant-folding, Orbit also performs beta-reductions (procedure invocations) wherever it is determined beneficial to do so, including those for primitive operations, which specify their own partial evaluators (this is discussed in more detail in section 9.1).

The optimizations currently performed are:

1. `((lambda () body)) => body`
2. `((lambda (x) (... x ...)) y) => (... y ...)` if `x` is referenced only once or if it appears that the reduced cost of accessing `y` offsets the increase in code size that results from duplicating `y`.

[7] Cells are analogous to names in Algol-68. They are also analogous to the use of a store in denotational semantics, in that either a cell or a store seems necessary for a proper treatment of assignment, whereas a simple environment is sufficient for "constants" and other variables which are bound but not subsequently altered.

3. Constant folding of primitive operations, including conditionals.
4. Substituting values that are defined in this module or in another module whose declaration file (see section 3.6) is being used in the compilation.
5. Removing unused arguments to procedures. Note that as a result of CPS conversion, arguments cannot be calls and thus cannot have side-effects.
6. Removing side-effect free calls whose results are not used.
7. Various optimizations specific to primitive operations, involving multiple value returns, mutually recursive definitions, etc.
8. Transformations of conditional expressions to effect "boolean short-circuiting;" i.e. the evaluation of a conditional expression for an *effect* rather than a value. More specifically:

   a. `(if (if a b c) d e) => (if a (if b d e) (if c d e))`
      But to avoid duplicating d and e the actual transformation is:
```
              (let ((x (lambda () d))
                    (y (lambda () e)))
                (if a (if b d (x)) (if c d (y))))
```
      The simpler version is used in most of the examples below.

   b. The results of tests are propagated:
```
              (if a (if a b c) d) => (if a b d)
              (if a b (if a c d)) => (if a b d)
```
      etc.

   c. `(if (let ((...)) a) b c) => (let ((...)) (if a b c))`

The following two examples demonstrate how the transformations on conditionals perform boolean short-circuiting. In the first example, not is defined as a procedure (and is exactly how it is defined by Orbit). A simple conditional expression using not is then simplified by the above transformations.

```
not == (lambda (x) (if x nil t))        ; Definition of the procedure NOT

(if (not x) 0 1)
   => (if (if x nil t) 0 1)             ; Substitute the definition of NOT
   => (if x (if nil 0 1) (if t 0 1))    ; (if (if ...) ...)
   => (if x 1 0)                        ; (if nil a b) => b, (if t a b) => a
```

The second example uses and and or, which are defined as macros by Orbit:

```
(and x y) => (if x y nil)               ; Definition of the macro AND
 (or x y) => (let ((p x)                ; Definition of the macro OR
                   (r (lambda () y)))
               (if p p (r)))
```

The definition of or is complicated by the desire not to evalute x twice and to avoid variable conflicts between the introduced variable p and the expression y. Now a detailed example:

```
(if (or (and x y) z) 0 1)
   => (if (let ((p (if x y nil))          ; Expand OR and AND
                (r (lambda () z)))
            (if p p (r)))
          0 1)

   => (let ((p (if x y nil)))             ; Move the LET out of the IF
        (if (if p p ((lambda () z)) 0 1))); and substitute R

   => (let ((p (if x y nil)))             ; (if (if ...) ...)
        (if p (if p 0 1) (if z 0 1)))
```

```
=> (let ((p (if x y nil)))              ; Propogate the result of (if p ...)
     (if p 0 (if z 0 1)))

=> (if (if x y nil) 0 (if z 0 1))       ; Substitute P (it occurs just once)
=> (let ((s (lambda () (if z 0 1))))    ; (if (if ...) ...)
     (if x (if y 0 (s)) (if nil 0 (s))))

=> (let ((s (lambda () (if z 0 1))))    ; (if nil a b) => a
     (if x (if y 0 (s)) (s)))
```

The final result of this example may look less than optimal. However, since calling a known procedure with no arguments compiles as a jump, the above code is actually equivalent to:

```
    if X then if Y then 0
                    else goto S
         else goto S
 S: if Z then 0
         else 1
```

which is the code one would hope for when compiling `(if (or (and x y) z) 0 1)`. Note that this code is produced without any special information about `and` or `or`. Any similar control construct that is translated into `if` will be compiled equally by the same general transformations.

These transformations are in most respects the same as those described in [14] and [2], but they are performed on CPS code instead of source code (the examples above were not shown in CPS only to make them easier to read). The organization and simplicity that CPS and assignment conversion bring to the intermediate code allows a few fairly simple transformations such as these to produce highly optimized code. Work in progress on more complex optimizations (such as type inference) indicates that the structure of the intermediate code will continue to be beneficial.

3.5. Other Transformations

As further examples of how the compiler deals with specific procedures we will show how multiple value returns and mutually recursive expressions are implemented. In our dialect of Scheme there are two procedures that deal with multiple value returns. `(return value1 ... valueN)` returns its arguments as multiple values, and `(receive-values receiver sender)` calls `receiver` on the values returned by `sender`, a procedure of no arguments. For example:

```
(receive-values (lambda (x y) (list x y))
                (lambda () (return 1 2)))       => (1 2)
```

Usually the macro `receive` is used instead of calling `receive-values` directly:

```
(receive <identifier-list>   =>  (receive-values (lambda <identifier-list>
         <call>                                    . <body>)
  . <body>)                                      (lambda () <call>))
```

The above example can then be written as:

```
(receive (x y)
         (return 1 2)
  (list x y))                           => (1 2)
```

Although `return` and `receive-values` cannot be written directly in Scheme, they do not require special handling in Orbit, since continuations can take multiple arguments. Their definitions include declarations of code transformations that can be used by the optimizing phase of the compiler to simplify calls to the procedures. The transformations are shown here in their CPS form (and thus note the `<cont>` argument):

```
(return <cont> <v1> ... <vN>)    =>    (<cont> <v1> ... <vn>)
(receive-values <cont> (lambda (k . <vars>) . <body>) <sender>)
   => (<sender> (lambda (<vars>)
                  ((lambda (k) . <body>) <cont>)))
```

Here is the result of applying these two transformations to the example shown above:

```
(receive-values (lambda (x y) (list x y))
                (lambda () (return 1 2)))
    => (receive-values <cont>       ; CPS conversion
                       (lambda (k1 x y) (list k1 x y))
                       (lambda (k2) (return k2 1 2)))

    => ((lambda (k2) (k2 1 2))      ; Transform RECEIVE-VALUES and RETURN
        (lambda (x y)
          ((lambda (k1) (list k1 x y))
           <cont>)))

    => ((lambda (x y)               ; Substitute for K1 and K2
          (list <cont> x y))
        1 2)
    => (list <cont> 1 2)            ; Substitute for X and Y
```

Mutually recursive expressions are defined using the macro `letrec`, which is expanded in two steps. First, any `<exp>`s that are not `lambda` expressions are handled using `set!`:

```
(letrec (...                 =>      (let ((x nil))
         (x (foo bar))                 (letrec (...
         ...)                                   ...)
  <body>)                                (set! x (foo bar))
                                         <body>)))
```

Once this is done the resulting expression is expanded into a call to the procedure Y:

```
(letrec ((v1 <exp1>)         =>   (Y N (lambda (v1 ... vn)
         ...                             (values (lambda () <body>)
         (vn <expN>))                            <exp1>
  <body>)                                        ...
                                                 <expN>)))
```

The procedure Y finds the least fixed point of its second argument, and is defined just as any other primitive operator with its own transformation and code generation methods. Thus unlike `return` and `receive-values`, its run-time definition can be written in Scheme:[8]

```
(lambda (n value-thunk)
  (let* ((values nil)
         (make-proc (lambda (i)
                      (lambda args
                        (apply (nth values i) args))))
         (procs (do ((i 0 (+ i 1))
                     (p '() (cons (make-proc i) p)))
                    ((>= i n) (reverse p)))))
    (receive (body . real-values)
             (apply value-thunk procs)
      (set! values real-values)
      (body))))
```

3.6. Declaration Generation

In addition to compiled code for a target machine, Orbit produces a compiled set of *declarations* that contain information about the compiled code that can be used in compiling other source modules. Such declarations include information about defined procedures, type information, and primitive operations. The declaration files are similar to "include" files in other languages. They allow the compiler to do type checking and optimizations between files.

[8]The similarity of Orbit's Y to the "paradoxical Y combinator" in the lambda calculus is not an accident.

The declaration files to be used in compiling a particular module are specified in a special form that appears at the beginning of the source file for the module. The definitions in the specified declaration files make up the compile-time environment for the module. By using the declaration files for the modules that will be present in the run-time environment, the run-time and compile-time environments can be made to correspond exactly.

4. Closure Strategy Analysis

Each LAMBDA-node in the CPS code tree will evaluate to a closure. The primary aim of *strategy analysis* is to determine where the environment for each closure will be allocated. The three possibilities we consider are described below.

Heap allocating a closure is the most general strategy, and must be used if the compiler cannot prove that a closure's lifetime is limited in a well-defined way. This is normally the case, for example, for a closure returned from a procedure call, or passed as a non-continuation argument to an unknown procedure. Although heap allocation is very general, it is relatively expensive, because the closure will not be reclaimed until a garbage collection occurs. Thus we generally try to allocate closures in one of the following two alternative ways.

There are many situations in which a closure's lifetime is limited in such a way that it may be allocated on the *stack*. There are two common cases where this is possible. The first is a closure representing the continuation to an unknown call; such a closure is unused once called. The second case is a closure representing a local recursion, where the compiler decides that keeping the environment in registers (another optimization - see below) would cause too much extra code to be generated. (The extra code results from the need to save and restore registers across a procedure call.)

Both of these cases, however, have associated difficulties, since the existence of `call-with-current-continuation` means that any continuation may actually have an unlimited lifetime. Fortunately, the compiler can generally ignore this fact, because the runtime system ensures that any continuation which is captured by `call-with-current-continuation` will be migrated dynamically to the heap.

In the case of a closure representing a local recursion, we must take care to remain properly tail-recursive. If the closure is allocated on the stack just before its code is jumped to, it must be removed when the recursion exits. In the case where the closure is always called with the same continuation (i.e., with the same stack pointer, as in a loop) this is not a problem because the distance from the stack pointer on exit to the stack pointer on entry is a constant known at compile time. But if we do not know at compile time which continuation the closure is called with, we do not know how far from the pushed closure we will be when the closure exits. In this case we generate runtime checks on exits from the closure to see if the pushed closure has been reached. If it has, it can be popped off. [12].

The third and most space-efficient way to allocate a closure is to spread its environment through the *registers*. This can be done if all places from which the closure might be called are known to the compiler. This is the case for (1) a `lambda`-expression in procedure call position, (2) a continuation for an open-coded operation, and (3) a closure bound to a `let` or `letrec` variable which is only called. Obviously in these cases the closure needs no code pointer because the compiler will know the address of the code when it is invoked. Thus no code is actually emitted when a `lambda`-expression meeting the requirements of this strategy is evaluated. Before the code is jumped to, the free variables are moved to wherever the register allocator has decided they should go. If good register allocation is done, little or no code will be generated at this time.

Even when it is possible to spread an environment in the registers, there are certain cases where it may not be desirable. For example, if a loop has an unknown call in its body, the environment in the registers would have to be saved as part of the continuation to the unknown call, and restored to the same registers when jumping to the top of the loop. Depending on the number of registers being saved and restored, this may be more expensive than allocating the closure using one of the other methods.

5. Closure Representation Analysis

Orbit works hard to reduce the amount of storage used by closures. Strategy analysis determines when a closure may be allocated on the stack or in registers, but even in the case that a closure must be heap-allocated, several techniques are employed to minimize the size of the runtime data structures representing the closure and its environment.

Orbit reduces heap allocation for closures by packing the code pointers for procedures which reference the same variables (or a subset of the same variables) into a single runtime structure. Such *closure hoisting*, where closures may be allocated before the node in the code tree where they appear, leads to the sharing of environment without the overhead of a linked environment structure.

A common case of closure hoisting is when all top-level procedures in a module are put in the same closure. Most Lisp implementations make a very strong distinction between top-level procedures and general closures, in order to eliminate one or two instructions and memory references on entry to such top-level procedures, and two or three extra words of storage in the representation of such procedures. Orbit does not distinguish such procedures or the global environment. The above optimizations happen as a result of the general techniques referred to in the previous paragraph.

Closure hoisting is also used to reduce the number of variables which are live at certain points. At a particular node in the code tree, there may be variables which are live only because they are referenced by a continuation which will be evaulated at some later time. If the continuation were evaluated ahead of time, these variables would no longer be live. The examples given in section 8 illustrate this technique. The rule Orbit uses is that a continuation should be evaluated as soon as the values of all its free variables are available, but not until it is known to be necessary; e.g., continuations should not be hoisted above conditional expressions.

The representation analysis phase deals explicitly with the fact that assigned lexical variables have had cells introduced to hold their values. Since the introduction of cells implies that lexical variables are never assigned, we need not worry about special allocation of such variables in closures. But, as an optimization, we try to avoid actually allocating separate heap storage for the introduced cells. If a simple analysis shows that a cell will only be referred to by code in one closure and that the only operations done on the cell are to fetch its contents or modify its contents, then that cell can be eliminated, the "address" of the closure slot serving as the cell.

Note that cells for variables which are part of a stack-allocated closure may not be collapsed, due to the possibility of the closure being captured by a call to `call-with-current-continuation`. Stack-allocated closures must be "read only" because `call-with-current-continuation` is implemented by making a copy of the stack, and cells must never be copied. In practice, this is not a problem in Scheme, since assignments are generally used only for global variables and to implement objects with state; the former are created infrequently, and the latter will usually be in the heap. Iteration variables are always updated by rebinding rather than assignment, so the continuations which contain them may be overwritten. An interesting corollary of this, of course, is that users of Orbit will be encouraged to write code in applicative style (i.e., without side-effects), since such code will compile better, and cons less, than non-applicative code!

6. Register Allocation and Code Generation

6.1. Description of Virtual Machine

The virtual machine that Orbit "sees" has a stack, a heap, and N registers. The registers are divided into two classes: *rootable* and *non-rootable*. When a garbage collection occurs, only those values in rootable registers will be examined. It is necessary to make this distinction in order to make it possible for a garbage collection to occur at any time.

There are also several registers which are used by convention in the standard call and return sequences. There is one register to hold the procedure, M registers to hold the arguments, and a register to hold a representation of the number of arguments. The standard calling sequence is that the address of the closure being called is put in the procedure register, the arguments in the standard argument registers, and the number of arguments is placed in the specified argument count register. The latter is used to do runtime checks of the number of arguments, if desired, and to tell n-ary procedures how many arguments they are receiving.

Despite this register usage convention, none of the registers are firmly reserved in any way. Furthermore, some of the argument registers will be in the memory of the real machine, depending on how many registers the real machine has.

6.2. Stack Usage

Orbit uses the stack in a slightly different way from conventional compilers. A conventional compiler would, for each procedure compiled, create temporary variables and allocate space on the stack to hold them. If using the "caller saves" convention, live variables would be saved on the stack each time a procedure is called. Orbit does not allocate "temporary" variables in stack frames. After CPS conversion there is no distinction made between variables that the programmer has introdeced and variables the compiler has introduced to hold the values of intermediate computations. All values live either in registers or in the environment of a closure. When a procedure is called non-tail-recursively, its continuation will be pushed on the stack and the code for the procedure will be jumped to. There is no notion of saved registers being restored; the continuation is a full closure and the "saved" variables have become part of its lexical environment.

In the event that many closures are allowed to keep their environments in registers, there is a considerable advantage to doing a good job of register allocation. Orbit optimizes register usage by virtually increasing the size of basic blocks (by ignoring splits - see section 6.4 below) and by allowing the value of a variable to exist in many machine locations at the same time. One major advantage of assignment conversion is that the code generator can freely assign any variable to an arbitrary number of locations.

Many register allocators do poorly on code containing references to *variables stored in closures*. The problem is that when a variable is fetched from its "home" in a closure and put in a register, the fact that the value of the variable is actually in that register is immediately "forgotten." If the same variable, or perhaps another in the same closure, is needed soon after, a series of environment pointers will have to be fetched again. Our solution is to use a *lazy display*. In a conventional display a dedicated set of registers holds all of the active environment pointers. However, we cannot afford to dedicate registers in that way, especially since some of the values held in the display may never be needed, depending on the dynamic behavior of the code. In a *lazy display* environment pointers are treated just like variables, being allocated to registers as they are needed; i.e., "by demand."

6.3. Code Generation

Most application nodes in the code tree are either procedure calls, returns (calls to continuations), or calls to primitive operations to be coded in-line. In the latter case the (usually machine-dependant) code generation routine for the primitive operation is invoked. Generating code for a procedure call or return involves doing a "parallel assignment" of the arguments, as described below. (For the remainder of this discussion procedure call means call or return.)

Before a procedure call, the arguments are in certain registers (or in environments) and they are to be bound to new locations corresponding to the formal paramaters of the procedure being called. If the compiler has information about which registers the arguments should be passed in (because it has already compiled the body of the procedure being called), it uses those registers. Otherwise it uses the registers given by the standard calling convention. When a procedure is called whose environment has been decided to be allocated in registers, the environment variables are treated as additional arguments. In the simplest case, all of the arguments are in registers and the arguments just have to be shuffled around to put them is the expected places. In general, however, things are much more complicated because arguments may be in arbitrary environments and some arguments may evaluate to closures which need to be consed. In any case, one hopes that the arguments which are the results of previous computations will have been targeted to the correct position so that moving these values to the expected places will not generate much code.

6.4. Join Points

To achieve a form of global register allocation, basic blocks are *extended* by attempting to ignore "splits" and "joins" in the code, where splits are represented as conditionals, and joins as procedures which are called only from known places. The compiler tries to determine heuristically which branches of conditionals are most likely to be taken at runtime, and the code generator chooses its path through the code accordingly. When it arrives at a join point for the first time, it ignores this fact, merely notating which registers hold which values and continuing as if it were compiling straight-line code. When compiling subsequent arrivals at the join point the registers must be "synchronized." (This overall strategy has been termed *trace scheduling* as it was initially used in the context of compacting horizontal microcode for parallel machines [5, 4].)

7. Integral Assembler

TC made use of system supplied assemblers, but we elected not to follow this route for several reasons. First, we found that a considerable effort was spent coding the backend to accommodate the short-comings of the assemblers. Second, we needed a representation of the instruction stream in order to do peephole optimizations; given that, assembly seemed like a small addition. Third, we could achieve a higher level of system integration by defining our own binary file format.

"Assembler" is actually a misnomer. Its input is a flow graph of basic blocks, each ending logically with a branch instruction. These branches do not actually appear with the other instructions in the block; rather, each block is marked with a branch condition (<, >, =, always, etc) and two destination blocks, selected on the result of the condition. In the course of its operation, the assembler first orders the blocks using heuristics intended to minimize the number and size of branch instructions. Once the block order is known, the assembler converts flow graph edges to branch instructions where needed.

The block-ordering heuristics are not good enough to avoid introducing an extra branch into loops, so the code generator provides a hint to the block ordering routines; namely, to order the head of the loop after the body. For simple loops this will result in a code sequence of <before loop>, <branch to test>, <body of loop>, <test>, <conditional branch to body>. That is, there is only one branch for loop control inside the loop.

The assembler is driven off a table of instruction descriptions for the target machine. An instruction is described as a parameterized sequence of bit fields. Each bit field has a width and a value, either of which may need to be determined in the process of assembling. The width of a field may be specified as a simple function of the distance between to points in the bit stream. The assembler finds the minimum possible width for such fields, which amounts to minimizing branch and pc-relative displacments [15].

The structure of the assembler has also been designed to support tail merging [16] and peep hole optimizations, though these have not yet been implemented.

8. Examples

As an example of the quality of Orbit's compiled code, figures 8-1 and 8-2 show the source, intermediate CPS, and VAX assembler code for a Fibonacci number generator. Several things are worth noting about the resulting code:

First, the environment for a local recursive procedure would normally be stack allocated, but FIB1 does not need an environment. Thus calls to FIB1 are simply jumps. Furthermore, the first call to FIB1 from C32 is just a "fall through," which is due to the fact that the code generator tells the assembler which jumps to a particular block of code are *back edges* in the program graph - the assembler orders the basic blocks of code accordingly. Lastly, the compiler hoists the consing of continuation C26 so that the value of X13 in register a1 can be overwritten with V19 and continuation C24 so that V25 can be overwritten by V21.

```
(define (fib x)
  (letrec (((fib1 x) (if (fx< x 2)
                         1
                         (fx+ (fib1 (fx- x 1))
                              (fib1 (fx- x 2))))))
    (fib1 x)))
```

Note: `fx+`, `fx-`, etc. are arithmetic operators on small integers.

```
FIB  = (lambda (K5 X6) (Y K5 '1 Y11))
Y11  =   (lambda (C9 FIB8) (C9 C10 FIB1))
C10  =     (lambda (K22) (FIB8 K22 X6))
FIB1 =     (lambda (K12 X13) (conditional C31 C32 FX< X13 '2))
C31  =       (lambda () (K12 '1))
C32  =       (lambda () (FX- C20 X13 '1))
C20  =         (lambda (V19) (FIB8 C26 V19))
C26  =           (lambda (V25) (FX- C22 X13 '2))
C22  =             (lambda (V21) (FIB8 C24 V21))
C24  =               (lambda (V23) (FX+ K12 V25 V23))
```

Figure 8-1: Source and CPS Code for Fibonacci Function

A second example is shown in figures 8-3 and 8-4, which is for a function `delq!` that destructively removes all instances of an object from a list. In this example code for the MC68000 has been generated. d5 is the number of arguments register. In Orbit, the two low-bit tags of a pair are 11 and the CDR comes before the CAR. Thus if a pair is in register `r`, `1(r)` accesses the CAR and `-3(r)` the CDR.

Orbit does not produce assembly files. Shown below is the code output by an auxilliary procedure for producing assembly listings. The .template field means that there is a descriptor of the code segment following which includes information such as the number of arguments the code expects, information used by the garbage collector, and debugging information. According to the standard convention, the first argument to a procedure is passed in a1 and the representation of the number of arguments is passed in s3. The register task always points to data such as the address of the code which does a procedure return.

```
D1716:   .template              ; descriptor for FIB
D1717:   jbr  FIB1              ; initial call to FIB
                                ; argument already in a1 because FIB will
                                ; be called by the standard convention
C32:     pushl   a1             ; the continuation refers to x so save it
         pushal  D1718          ; push address of continuation C26
         subl2   $4,a1          ; (- x 1) and jump to FIB1 which has
                                ; been assembled to the next instruction
FIB1:    cmpl    a1,$8          ; compare x,2 (with 00 tag field in low bits)
         jge     C32
C31:     movl    $4,a1          ; return 1 (with 00 tag field in low bits)
         mnegl   $2,s3          ; indicate 1 value returned
         jmp     *-36(task)     ; jump to return code
D1722:   .template              ; descriptor for C26
                                ; V25 is returned in a1
D1718:   pushl   a1             ; save V25 = (fib1 (- x 1))
         pushal  D1723          ; push address of continuation C24
         subl3   $8,12(sp),a1   ; compute (- x 2)
         jbr     FIB1           ; call FIB1

D1724:   .template              ; descriptor for C24
                                ; V23 is returned in a1
D1723:   addl2   4(a7),a1       ; add results of 2 calls to FIB1
         addl2   $16,sp         ; restore continuation register
         mnegl   $2,s3          ; one value returned
         jmp     *-36(task)     ; jump to return code
```

Figure 8-2: VAX Assembler Code for Fibonacci Function

One interesting thing about this example is that it shows global register allocation in a recursion. Since none of the calls to DELQ! occur from inside a real runtime closure, the environment consisting of just OBJ1 has been allocated to register a2.

9. Miscellaneous

9.1. Primitive Operations

Rather than bury the details of how to compile primitive operations in the internal logic of the compiler, they are implemented by specifying their behavior in Scheme source modules. This enhances modularity and portability by facilitating adjustments to the compiler's source and target languages. The specifications of primitives are accomplished with a special form (primitive <body>), where <body> specifies not only the primitive's compile-time behavior (such as code generation and optimization strategies), but also its run-time behavior (i.e., how it should behave if it were called dynamically at run-time).

9.2. Runtime Environment

Traditional Lisp (and Scheme) implementations require that a compiled program be loaded into a substantial Lisp runtime environment. Usually, not all of this environment is required for the execution of the program. While

```
(define (delq! obj list)
  (letrec ((delq! (lambda (list)
                    (cond ((null? list) '())
                          ((eq? obj (car list)) (delq! (cdr list)))
                          (else (set-cdr! list (delq! (cdr list)))
                               list)))))
    (delq! list)))

DELQ = (lambda (K0 OBJ1 LIST2) (Y Y14))
Y14  =   (lambda (C12 DELQ!3) (C12 C13 P17))
C13  =     (lambda () (DELQ!3 K0 LIST2))
P17  =     (lambda (K4 LIST5) (conditional C20 C21 EQ? LIST5 '()))
C20  =       (lambda () (K4 '()))
C21  =       (lambda () (car C45 LIST5))
C45  =         (lambda (V44) (conditional C24 C27 EQ? OBJ1 V44))
C24  =           (lambda () (cdr C26 LIST5))
C26  =             (lambda (V25) (DELQ!3 K4 V25))
C27  =           (lambda () (cdr C37 LIST5))
C37  =             (lambda (V36) (DELQ!3 C39 V36))
C39  =               (lambda (set38) (set-cdr! B35 SET38 LIST5))
B35  =                 (lambda (IGNORE34) (K4 LIST5))
```

Figure 8-3: Source and CPS Code for DELQ!

```
D104:  .template
                             ; LIST5 wants to be in a1
D105:  exg a2,a1             ; OBJ1 kept in a2
       jbra P17              ; jump to DELQ!
C21:   cmpa.l 1(a1),a2       ; (eq? obj (car list))
       jbeq C24
C27:   move.l a1,-(sp)       ; save value of LIST5
       pea D107              ; push address of continuation C39
       movea.l -3(a1),a1     ; arrgument to DELQ! is (cdr list)
       jbra P17              ; jump to DELQ!
C24:   movea.l -3(a1),a1     ; argument to DELQ! is (cdr list)
P17:   cmpa.l d7,a1          ; (null? LIST5) null is always in d7
       jbne C21
C20:   movea.l d7,a1         ; return '()
       moveq #-2,d5          ; indicate 1 value returned
       jmp -40(task)         ; jump to return code

D112:  .template             ; descriptor for C39
D107:  movea.l 4(sp),a0      ; get LIST5
       move.l a1,-3(a0)      ; set CDR to be value returned by DELQ!
       movea.l a0,a1         ; return LIST5
       lea 8(sp),sp          ; restore continuation
       moveq #-2,d5          ; indicate 1 value returned
       jmp -40(task)         ; jump to return code
```

Figure 8-4: MC68000 Assembler Code for DELQ!

there is a complete Scheme runtime environment known as the *T system*[9] into which Orbit's object files may be loaded, an object file generated by Orbit may alternatively be linked with other object files to generate an executable image, in the same manner as C or Modula-II programs. Normally users will want to run programs in the context of some or all of the T system, which includes a storage manager (garbage collector), an I/O subsystem, various utility modules (such as hash tables), support for multiprocessing (tasks), parser, interpreter, dynamic

[9]T is the overall name given to the Scheme programming environment project of which the Orbit compiler is a component; see [10, 11].

loader, debugging tools, text editor (eventually), and the compiler itself. But independent execution may be desirable because of address space limitations in a cross-development context, or simply to decrease image startup costs within an operating system not devoted to executing Scheme programs. We anticipate that in conjunction with a compiler of Orbit's quality, this feature will overcome much of the traditional resistance to Lisp in the area of systems programming.

The T system relies heavily on Orbit's ability to compile calls to *foreign procedures*, that is, procedures not written in Lisp, and compiled by another compiler. Orbit's foreign procedure call mechanism allows the user to specify both the linkage used by a particular foreign procedure and the conversions necessary between Scheme's data representations and foreign data representations.

10. Conclusions

The combination of lexical scoping, full closures, and first-class continuations creates a unique and challenging task for compiler designers. We have found that the general CPS approach induces a particular style of compiler writing that has many benefits not originally identified by Steele, and results in a compiler that not only has a simple and modular organization, but also generates very efficient code.

11. Acknowledgements

We wish to thank the Department of Computer Science at Yale (and John O'Donnell in particular) for its early support of the T project, in which Orbit has its roots. We also gratefully acknowledge the support of Forest Baskett at the DEC Western Research Laboratory.

References

1. Abelson, H., Sussman, G.J. and Sussman, J.. *Structure and Interpretation of Computer Programs.* MIT Press and McGraw-Hill, 1985.

2. Brooks, R.A., Gabriel, R.P. and Steele, G.J. Jr. An optimizing compiler for lexically scoped LISP. Proc. Sym. on Compiler Construction, ACM, SIGPLAN Notices 17(6), June, 1982, pp. 261-275.

3. Clinger, W., et al. The Revised Revised Report on Scheme, or An UnCommon Lisp. AI Memo 848, Massachusetts Institute of Technology, Aug., 1985.

4. Ellis, J.R. *Bulldog: A Compiler for VLIW Architectures.* Ph.D. Th., Yale Univ., 1985. available as Research Report YALEU/DCS/RR-364.

5. Fisher, J.A. Very Long Word Architectures. YALEU/DCS/RR-253, Yale University, Dec., 1982.

6. Friedman, D.P. and Haynes, C.T. Constraining control. 12th ACM Sym. on Prin. of Prog. Lang., ACM, 1985, pp. 245-254.

7. Gabriel, R.P.. *Performance and Evaluation of Lisp Systems.* MIT Press, Cambridge, Mass., 1985.

8. Haynes, C.T., Friedman, D.P. and Wand, M. Continuations and coroutines. Sym. on LISP and Functional Programming, ACM, Aug., 1984, pp. 293-298.

9. Powell, M.L. A portable optimizing compiler for Modula-2. Proc. Sym. on Compiler Construction, ACM, SIGPLAN Notices 19(6), June, 1984, pp. 310-318.

10. Rees, J.A. and Adams, N.I. T: a dialect of LISP or, Lambda: the ultimate software tool. Sym. on Lisp and Functional Prog., ACM, Aug., 1982, pp. 114-122.

11. Rees, J.A., Adams, N.I. and Meehan, J.R. The T Manual. 4th edition, Yale University, Jan., 1984.

12. Rozas, W. (personal communication).

13. Steele, G.L. Jr. and Sussman, G.J. The Revised Report on Scheme. AI Memo 452, MIT, Jan., 1978.

14. Steele, G.L. Jr. RABBIT: a compiler for SCHEME. AI Memo 474, MIT, May, 1978.

15. Szymanski. "Assembling code for machines with span-dependent instructions". *CACM 21*, 4 (April 1978), 300-308.

16. Wulf, W., Johnson, R., Weinstock, C., Hobbs, S. and Geschke, C.. *The Design of an Optimizing Compiler.* American Elsevier, 1975.