ABSTRACT

ORBIT: An Optimizing Compiler For Scheme

David Andrew Kranz

Yale University

1988

It has often been assumed that the performance of languages with first-class procedures is necessarily inferior to that of more traditional languages. Both experience and benchmarks appear to support this assumption. This work shows that the performance penalty is only a result of applying conventional compiler technologies to the compilation of higher order languages. These technologies do not adapt well to the situation in which closures of unlimited extent can be created dynamically.

The ORBIT compiler is based on a *continuation-passing model* instead of the traditional procedure call/return. The problem of reducing heap storage is solved using new algorithms for *closure analysis*, allowing many objects to be allocated on a stack or, better still, in machine registers. *Closure packing and hoisting* allow more than one procedure to share an environment without introducing indirection. Move instructions and memory references are reduced by passing arguments in registers and using a dynamic register allocation strategy. Register allocation and code generation are accomplished at the same time, with environment pointers being treated as variables. Environment pointers are kept in a *lazy display*, being brought into registers and cached when needed. The interaction of this strategy with the closure analysis also allows many optimizations based on type information to be performed.

Benchmarks are presented to show that, using these new techniques, the performance of programs written in higher order languages almost equals that of programs written in Pascal in both space and time. Thus the greater expressive power of higher order languages and debugging ease of traditional LISP systems need not be sacrificed to attain good performance.

ACKNOWLEDGEMENTS

I would like to thank the many people who helped me over the years. Foremost were the other members of the T project: Norman Adams, Richard Kelsey, Jim Philbin and Jonathan Rees. My advisor Paul Hudak gave me constant encouragement. I had many helpful discussions with Jonathan Young, Ben Goldberg, Kai Li, my readers Alan Perlis and Will Clinger, Rick Mohr and John Ellis.

Contents

Chapter 1

Compiling Scheme

The programming language Scheme is much more expressive and general than traditional languages such as Pascal. In spite of this fact, it is possible to compile Scheme programs as efficiently as those written in Pascal.

1.1 Introduction

Scheme [Steele 78, Clinger 85] has been around for a long time. It is an expressive language which is easy to program in, and is often used in programming and data-structure courses [Abelson 85]. Nevertheless, Scheme has never been accepted as a "real" programming language because even moderately efficient implementations did not exist compared with other, more widely used languages such as Pascal or C.

In his seminal Rabbit paper [Steele 78b], Steele recognized that a certain model of compilation, based on "continuation-passing" instead of procedure call and return, was a good model for compiling Scheme. This dissertation examines the issues of code generation and efficient allocation of closures that Rabbit did not address. New techniques are used to build a production quality compiler for Scheme which is as efficient as a Pascal compiler. In fact, it turns out that for a large class of programs the performance is better than Pascal, indicating that the techniques used might be applied to the compilation of more traditional languages.

1.2 Scheme

Scheme is a dialect of Lisp with several important properties:

1. It is lexically scoped.
2. It has first-class procedures and continuations.
3. It is tail-recursive.

Unlike Lisp, the procedure call position is evaluated along with all of the arguments and in exactly the same way. Also, a lambda expression *evaluates* to a procedure. When the procedure is called, the values of the variables free in the lambda expression are the same as when the lambda was evaluated – this is just what lexical scoping means. Procedures are first-class objects, meaning they can be used freely in arbitrary contexts, thus providing an expressive power that is absent in traditional Lisps. On the other hand, they also provide the implementation with storage management problems (the "funarg" problem) that are absent in traditional Lisp implementations: a procedure must contain both *code* and an *environment*. Procedures are often referred to as *closures*. In addition, first-class continuations can be used to build interesting control constructs [Wand 80, Haynes 84, Haynes 84b].

It has long been known that an iteration is equivalent to a tail-recursion. To say that a language such as Scheme has the property of tail-recursion is simply to say that an algorithm expressed as a tail-recursion is *guaranteed by the implementation to behave in the same way* as if it had been expressed iteratively. These behaviors are normally different in traditional languages. In Scheme there are no primitive iterative constructs like **for** or **do**. Iterative syntax may be used in a program, but it is just syntactic sugar and expands into recursive procedure calls.

1.3 Compiling Scheme Efficiently

Why is Scheme difficult to compile? This question is partially answered by saying that it is the same reason as for any language: code generation and register allocation are difficult problems. A Scheme compiler must deal with these issues, but must also worry about the fact that procedures are not merely code addresses, as in traditional languages, but real data objects which must be properly allocated somewhere in the machine. Further problems are caused by the fact that Scheme is dynamically typed, linked, and garbage collected. Dynamic typing and garbage collection imply that the type of runtime objects must be identifiable at all times. This overhead must be reduced by taking advantage of type information provided by the programmer and/or inferred by the compiler.

The important thing to recognize is that Pascal and C are really just restricted subsets of Scheme, in particular those parts of Scheme which can be efficiently implemented using conventional compiler technology! It seems reasonable that a Scheme program that uses only the restricted subset should have the same performance as the same program written in Pascal. What is needed is a "pay as you go" implementation; the most general language constructs should be implemented as efficiently as possible, but should not reduce the efficiency of less general constructs. In other words, a simple loop written in Scheme should not run more slowly than it would in Pascal just because Scheme allows first-class procedures and continuations.

Implementations of languages with first-class procedures have been approached from two completely different angles. Common Lisp [Steele 84] allows first-class procedures and there have been successful compilers written for it [Brooks 86]. These efforts used conventional compilation techniques primarily, with an implementation of closures added on. The result is that traditional constructs are implemented quite efficiently, but closures are not implemented nearly as efficiently as possible.

The other approach is to treat closures as fundamental, with loops and other simple constructs being particular cases. This is the approach suggested by Rabbit and is the one described in the succeeding chapters. The compiler knows nothing about particular kinds of loops, and it will be seen that many optimizations traditionally performed in and around loops apply to general recursions as well.

1.4 Orbit

The compiler described in this thesis is called Orbit [Kranz 86] and it is written in a dialect of Scheme called T [Rees 82, Rees 84, Slade 87]. Orbit runs in the latest implementation of T called T3. Orbit is a replacement for the old T compiler TC, which was based on [Brooks 82]. TC is another example of a compiler that used conventional methods and added on closure handling.

The rest of the dissertation is organized as follows:

- Chapter 2 explains how Orbit is influenced by the T system.
- Chapter 3 covers the front end of the compiler.
- Chapter 4 details the algorithms for doing *closure analysis*.
- Chapter 5 details the strategies for register allocation and code generation.
- In Chapter 6 some real examples of code are presented together with the results of benchmarks.

Chapter 2

Preliminaries

A compiler for a dynamically typed and linked language such as Lisp or Scheme must concern itself with generating code that satisfies the constraints of a large runtime system. This chapter will describe those constraints for the T system and Orbit, ending with an overview of various phases of the compiler.

2.1 Garbage Collection

Most of the constraints imposed on the compiler by the runtime system are due to the garbage collector. The T3 system and Orbit are meant to support lightweight processes. Because correctly polling for interrupts seemed more expensive, we have imposed the requirement that the system be able to handle an interrupt between any two instructions, unless they are in critical sections.

On the other hand, it is often necessary to manipulate non-tagged data. Even if using non-tagged data is not necessary, the compiler may elect to do so for reasons of efficiency. When a garbage collection occurs, it must be possible to identify all locations in the machine which contain tagged objects. These locations are said to be *rootable*, while all other locations are *non-rootable*. To satisfy that condition, the registers are partitioned into two classes, one to hold rootable data, and the other non-rootable. The compiler must enforce the partition at all times, locking out interrupts in some places if necessary. The data representations for Scheme objects must also be set up so that given a pointer to an object in memory, the rootable and non-rootable components can be determined. The overall structure of the tags and data representations are due to Jonathon Rees [Rees].

2.2 Calling Sequence

In the T3 system, machine resources are divided into three classes: a *heap*, a *stack*, and *registers.* The way Scheme objects are laid out is the same for both the heap and the stack. The only difference is that the heap is maintained by the garbage collector, while the stack is managed by the compiler. By registers, we mean machine locations which can be addressed by name. It is assumed that at least some of these registers are implemented in faster logic than the heap or stack.

The T3 system, like any other, has a *standard calling sequence* for passing arguments when a procedure call occurs. This calling sequence is always used if the compiler does not know that a different parameter passing mechanism is expected by the procedure being called. In T3, the arguments are passed in registers. If nothing is known about the procedure being called, the arguments must be tagged objects, and hence must be passed in rootable registers. The rootable machine registers are referred to as P and A1,A2,...AN. The P register contains the procedure being called (code and environment), A1 contains the first argument, and so on. The register AN is not used in the standard calling sequence because it is needed for other purposes. If there are more than $N - 1$ arguments, the remainder are passed in a set of memory locations at an address known by the compiler. This set of memory locations is treated as a global resource, just like the machine registers are.

2.3 Primitive Operators

Orbit achieves a degree of portability since it has a very small number of machine-dependent primitive operators. These operators fall into several classes:

- ALU operations such as addition, subtraction, etc.
- References and assignments to locations in Scheme objects. Examples include: structure references, car and cdr, and vector references. All of these constructs are handled by the same code generation routines.
- Special operators which are not part of Scheme but are used by the garbage collector and debugger.

These primitive operators are defined using the **primop** special form. A primop form evaluates to a Scheme object which can handle messages about code generation, type, or anything else the compiler wants to know about. As an example:

```
(define-constant fixnum-add
  (primop fixnum-add ()
    ((primop.generate self node)
     (generate-fixnum-binop node 'add))
    ((primop.simplify self node)
     (simplify-fixnum-add node))
    ((primop.type self node)
     '#[type (proc #f (proc #f fixnum) fixnum fixnum)])))
```

Unlike most LISP systems, the interpreter and compiler share the same definition of fixnum-add. When this form is compiled two things happen:

1. The compiler will be able to associate the variable fixnum-add with a code generation procedure, a simplification procedure, and type information.

2. A procedure is created that adds two fixnums and does type checking. In the runtime system, this procedure will be bound to **fixnum-add**.

2.4 Overview

The compilation process in Orbit is divided into a number of phases, listed below in the order in which they take place:

Alpha conversion creates a new version of the source program in which all macros have been expanded and all bound variables have been given unique names.

CPS conversion makes continuations explicit by transforming the program to *continuation passing style*. Each lambda expression is given an additional formal parameter bound to a continuation to be invoked with the result of executing the body. Each call is provided with a corresponding extra argument, the continuation to the call. An intermediate node tree is produced.

Assignment conversion eliminates assignments to bound variables by a transformation on the node tree. Explicit locations are introduced to hold the values of assigned variables bound by a lambda.

Early binding incorporates information from other compiled modules into the current compilation.

Program transformations either reduce the size of the node tree or simplify the analysis and code generation phases.

Live variable analysis determines the set of variables live at each lambda in the node tree.

Closure strategy analysis determines where each closure will be allocated at runtime and how each closure will be called.

Representation analysis determines the actual structure of each closure and how variables with a known type will be represented at runtime.

Register allocation and code generation are done in the same phase.

Assembly takes all of the instructions generated by the previous phase and produces a code vector.

The compiler was written by several people. Richard Kelsey wrote the phases up to and including early binding, and will be described in [Kelsey]. Norman Adams wrote the assembler. I was responsible for the rest; most of the following chapters will concentrate on closure analysis and code generation.

Chapter 3

The Front End

The front end of Orbit converts the source program into an intermediate node tree and then performs a series of passes over that tree. Each pass gathers information and annotates the tree, possibly transforming it. The annotations are used in later passes.

Conceptually, the compiler accepts three arguments:

1. A source expression in the form of a list of Scheme objects. The process of converting a source program as a string of characters into such a list will have already occurred.

2. A *syntax table*, that maps *reserved words* to *syntax descriptors*. A reserved word is a symbol that, when appearing in the call position of an S-expression, will be interpreted according to the syntax descriptor instead of as a procedure call. A reserved word will be either a *macro*, that will cause a source to source transformation, or syntax primitive to the compiler, such as IF.

3. An *early binding environment* that maps symbols to information the compiler can use.

The first pass in the compilation is *alpha conversion*. Lexical variables in different scopes, but with the same name, are renamed with unique identifiers. In addition macro-expansion is performed. The syntax table is consulted to determine which forms need to be macro-expanded. The early binding environment is consulted for available information about variables free in the source code, often referred to as global variables. For example, information about + might indicate e.g., that (+ 1 2) could be replaced by 3 during the code transformation phase.

3.1 CPS conversion

Conversion to *continuation passing style* is a program transformation, apparently due to van Wijngaarten. The following quote attributed to him [Felleisen 87]:

> ... this implementation [of procedures] is only so difficult because you have to take care of the goto statement. However, if you do this trick I devised, then you will find that the actual execution of the program is equivalent to a set of statements; no procedure ever returns because it always calls for another one before it ends, and all of the ends of all the procedures will be at the end of the program: one million or two million ends. If one procedure gets to the end, that is the end of all; therefore, you can stop. That means you can make the procedure implementation so that it does not bother to enable the procedure return. That is the whole difficulty with procedure implementation. That's why this is so simple; it's exactly the same as a goto, only called in other words.

CPS conversion was first used in the context of a "real" compiler by Steele in his Rabbit Compiler [Steele 78b]. The transformation consists of adding an extra argument, a *continuation*, to each combination. Each lambda expression is similarly changed by introducing an extra formal parameter to correspond to the continuation it will be passed. A complete algorithm for this transformation is provided in Rabbit, and for Orbit will be detailed in [Kelsey]. After the transformation the notions of procedure call and return have been unified; i.e., procedures no longer return. Instead, a procedure does a computation and calls its continuation with the result of the computation as argument. Note that a continuation never receives a continuation as an argument.

As a simple example:

```
(define foo
  (lambda (x y)
    (+ (f x) (g y))))
```

might be transformed into

```
(define foo
  (lambda (k x y)              ;;; the procedure
    (g (lambda (v)             ;;; continuation #1
         (f (lambda (w)        ;;; continuation #2
              (+ k w v))
            x))
       y))
```

In this example **k** is the variable introduced to hold the continuation passed to **foo**. This procedure calls **g** with arguments **y** and continuation #1. Eventually **g** calls its continuation, causing its result to be bound to **v**. Then **f** is called with arguments **x** and continuation #2, the result of this call being bound to **w**. Finally, + is called with the original continuation **k**, causing **v** and **w** to be added together and the result passed to **k**.

3.1.1 Consequences of CPS Conversion

CPS conversion makes the compilation process easier in a number of ways. First, the variables that are bound by continuations correspond exactly to compiler "temporary variables", except that in Orbit there is no distinction made between these variables and variables introduced by the programmer. In addition, there is no longer the restriction that a procedure return a single value. Continuations can be generalized to receive any number of arguments just like any other closure. Another benefit of CPS conversion is that tail-recursion is automatically uncovered and made explicit; a tail-recursive call (more properly a "tail-call") is one in which the continuation is a variable, like the call to + in the example but unlike the calls to **f** and **g** where the continuation is a lambda expression.

In fact, CPS code is perfectly correct Scheme code. All program transformations done by Orbit are performed on the CPS code, which makes those transformations simpler to write due to the regular structure of CPS code. The result of CPS conversion is an intermediate node tree with only four kinds of nodes: lambda-nodes, call-nodes, reference-nodes, and constant-nodes. Call-nodes and lambda-nodes simply correspond to calls and lambdas in the CPS converted expression, while reference-nodes correspond to variable references. Lambda-nodes that result from CPS conversion are treated no differently from those that appeared in the source code.

There are two kinds of constant-nodes:

- Literal-nodes represent literal values such as numbers and strings.
- Primop-nodes represent references to "primitive operators" for which the compiler has information, and only appear in call position. These objects are injected into the node tree in two ways:
  - as a result of primitive syntax in the source expression, such as an IF or SET!. The compiler has built-in information about these kinds of constructs.
  - as the replacement for a reference-node to a free variable because of information about the free variable in the early-binding environment. For example, a reference-node for the variable + might be replaced

by a primop-node that tells the compiler how to generate in-line code for +.

Note that there is nothing in the node tree like a "while-node" such as might be found in conventional compilers. All iterative constructs in Scheme are actually macros that expand into recursive function calls.

As a result of CPS conversion, the node tree has two important properties:

- *Each child of a call-node is either a lambda-node, a reference-node, or a constant-node; i.e., the subforms of a call cannot be combinations.*
- *A lambda-node has exactly one child: the call-node corresponding to the body of the lambda.*

These properties give the node tree a regular structure particularly easy to work with.

There is a strong similarity between the CPS code and the "triples" that many conventional compiler use as their intermediate representation. The primary difference is that in CPS code continuations are explicit.

Another important consequence of CPS conversion is that the order of evaluation of arguments to combinations is made explicit. In the standard semantics of Scheme the order of evaluation of arguments is purposely left undefined, so the compiler has complete freedom to choose an order. The heuristics used by Orbit to choose an order will be discussed at the end of Chapter 5 (Register Allocation and Code Generation).

3.1.2 CPS Algorithm

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 that 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 that 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 [Steele 78]. A new variable **j** is introduced to avoid copying the continuation. The result is:

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

3.2 Assignment Conversion

The presence of a general assignment operator causes difficulties in all phases of the compiler because assignment implies that variables denote locations instead of values. It is easier for a compiler to assume that variables denote values because, if so, beta-substitution can be performed freely in program transformations. In addition, a variable (value) could then exist simultaneously in registers and the environments of any number of closures without complicating the assignment operation.

Orbit eliminates the assignment problem by doing a transformation on the node tree which eliminates all assignments to bound variables. This transformation is called *assignment conversion* and is illustrated by the following example, shown before CPS conversion:

```
(lambda (x)
   ... x ...
   ... (set! x value) ... )
```

which is then transformed into

```
(lambda (x)
  (let ((x' (make-cell x)))
     ... (contents x') ...
     ... (set-contents x' value) ...))
```

A new variable **x'** has been introduced which is bound to a *cell* (created by **make-cell**) containing the value of **x**. The operation **contents** fetches the value in the cell and **set-contents** modifies the value. The important result of the transformation is that there are no longer any assignments to the variable **x**. This transformation is applied to all bound variables which might be assigned.

Because Scheme is lexically scoped it is possible to identify all such variables.

Assignment conversion does simplify the compiler considerably, but there is a cost: in general, storage will have to be allocated for the cells and there will be an extra indirection to reference the value inside a cell. This problem will be addressed in Chapter 4 (Closure Analysis), but it should be noted for now that good Scheme programming style discourages the use of assignment.

Unbound (free) variables are treated differently. Because the compiler does not know all the places they might be referenced, it must assume that they could be assigned, and because not all references to free variables are known at compile-time (due to incremental compilation), their cells can not be eliminated. However, most free variables in a typical Scheme program are bound to constant procedures which would only be changed by *incremental redefinition*, which is a debugging feature supported by most Scheme implementations. Because incremental redefinition is a debugging feature, we accept the fact that it will be expensive, and thus assignment conversion is not performed on free variables. Instead, the runtime system is given the burden of finding all places where the value exists and changing them when an assignment occurs! Making sure all places can be found at runtime puts some constraints on the compiler which will be explained in a later chapter, but the constraints are accepted to avoid having cells for every free variable.

3.3 Program Transformations and Early Binding

The last phase in the "front end" is a series of program transformations, or "optimizations", to simplify the node tree. Some transformations are "built-in" to the compiler, such as:

1. **((lambda () body))** $\Rightarrow$ **body**.

2. **((lambda (x) (... x ...)) exp)** ⇒ **(... exp ...)** if **x** is referenced only once or if it appears that the reduced cost of accessing **exp** offsets the increase in code size that results from duplicating **exp**.

3. Constant folding of primitive operations, including conditionals.

4. Removing unused arguments to procedures. Note that as a result of CPS conversion, arguments cannot be calls and so cannot have side-effects.

5. Removing side-effect-free calls when the results are not used.

6. "Boolean short-circuiting" of conditional expressions.

Many other transformations are not specified in the compiler, but rather are contained in the early binding environment. For example, the symbol $+$ is associated with information that allows constant folding for addition to be performed.

Early binding environments work much like "include files" in other languages. The difference is that the "include file" is generated by the compiler. The result of compiling a module is not only a compiled-code object, but also an object containing information which could be used in compiling other modules. As an example, in one module the programmer might declare some variable to have a constant value. If the early binding environment of another compilation were augmented with that information, then during the transformation phase references to the variable would be replaced by that constant value. The early binding environment may contain many different kinds of information, including:

- A variable that has a constant value.

- A variable that is bound to a procedure where a call to the procedure may be *integrated*; i.e., replaced by the body of the procedure with actual arguments substituted for formal parameters.

- Type information about procedures.

- Any other (possibly machine-dependent) information that might be used by the compiler.

Chapter 4

Closure Analysis

During *closure analysis* Orbit chooses run-time representations for the closures corresponding to each lambda-node in the node tree. In previous phases, we have been concerned only with the abstract properties of a procedure, such as the type of each argument or whether the procedure accepts a variable number of arguments. It is crucial to represent lambda-nodes efficiently because the result of CPS conversion is a node tree with a very large number of such nodes. Closure analysis has two parts: *closure strategy analysis* determines where in the machine to allocate each closure, and *representation analysis* determines the actual structure of each closure at runtime.

4.1 Storage Management

A compiler for any language has to address the problem of storage management. Typically, in languages such as C or Pascal, this involves allocating static storage for declared arrays and structures; procedures are simply addresses. A mechanism for the dynamic allocation of objects must be present, but it can be very simple because the difference between the abstraction of an array or structure is not very different from the concrete representation, i.e. an aggregate with constant-time access to components.

Dynamically created procedures in a lexically scoped language are much more difficult. The ability to create such procedures exists in Pascal, but in a limited way: a procedure can be passed as an argument, but it cannot be returned as a value, stored into a data structure or assigned to a variable. That is, a procedural parameter may only be called. Even in with this restriction, generating efficient code for these kinds of procedures is difficult. Implementation of this language feature is normally "thrown in" in an inefficient way with the expectation that it will not be used very much, and certainly not if efficient code is a concern.

We believe that in order to generate good code for dynamically created procedures, it is necessary to treat them as the fundamental objects to be examined by the compiler. The storage management problem is to determine where the code-pointer and environment (closure) for each procedure will be stored at run-time. For the machines being considered, the three possibilities are the heap, the stack, and spread through the registers.

Allocating procedures in the heap is undesirable because even after the procedure becomes inaccessible, the storage will not be reclaimed until a garbage collection occurs. It is also the case that garbage collection is much more expensive than reclaiming storage from the stack or registers. Furthermore, in order to access the values of variables within a heap-allocated closure, a pointer to the closure must be accessible as long as any of the variables are live. On the positive side, heap-allocation is the most general strategy in the sense that any closure can be heap-allocated.

Stack-allocating a procedure is better, because it can be cheaply reclaimed by emitting code to pop it off the stack as soon as it is known to be inaccessible. In fact, in order for the implementation to remain properly tail-recursive, it *must* be reclaimed as soon as possible. Enforcing proper tail-recursion presents problems which will be addressed in another section. A further advantage of stack allocation is that in some cases the stack-pointer can be used to reference values in the closure.

It is best to keep values in a closure in registers. Register space does not need to be reclaimed, and accessing a value in the closure does not generate a memory reference. The register strategy is quite different from the other two as to *when* the cost of allocation is incurred. When a lambda-expression is *evaluated* to produce a closure that is allocated in memory (heap or stack), the storage must be allocated and the values must be moved into the closure. Calling such a closure is inexpensive because the values of the free variables are packaged up.

Allocating an environment to registers is equivalent to treating the free variables as if they were arguments. Evaluating a lambda-expression to produce a register-allocated closure does not cause any code to be generated. However, when such a closure is *called*, the values of the free variables must be moved into the registers where they have been assigned, just like the arguments to any closure. Since a closure will normally be invoked many times, it is important that a call to a register-allocated closure be inexpensive. It is up to the register allocation and code generation phases to ensure that the number of moves generated is minimized.

4.1.1 Closures at Runtime

All full closures, whether in the heap or on the stack, are represented as contiguous chunks of memory. The first word in a closure is a code pointer that has one or possibly two other functions:

1. It points to information such as:
   - The number of arguments expected.
   - Whether or not a variable number of arguments are allowed.
   - How to get to debugging information about the closure.
   - How to garbage collect the closure.
   - How to invoke a generic operation on the closure.
2. If the expression evaluating to the closure has free variables, some of which are global, it may point to the environment where the global variables are

stored.

The way in which one closure inherits variables from a superior closure depends on the relative strategies used to implement both. A heap-allocated closure cannot inherit variables from a stack-allocated closure because, in general, pointers from the heap into the stack are not allowed. In this case the reason the inferior closure was heap-allocated in the first place was that it was known that it might outlive the context of the stack-allocated superior. If variables are inherited from a heap closure they may be copied into the inferior closure or a pointer to the superior may be kept. On the other hand, when one stack-allocated closure is inheriting from another, their relative locations are fixed permanently so that the pointer to the superior is not needed. The pointer which is used to get variables from the inferior closure can be used for the superior as well.

If any free variables are to be accessed from a superior closure, a pointer to the superior will be stored in the second word. The remaining locations in the closure contain variable values and possibly additional code pointers. Allowing multiple code pointers in a single run-time structure allows *both* environment sharing and the fastest access to variables. As an illustration consider the following code:

```
(define (f a b)
  (lambda (c d)                              ;;; lambda #0
    (list (lambda () (+ a b c d))            ;;; lambda #1
          (lambda () (* a b c d)))))         ;;; lambda #2
```

The closures, all in the heap, could look like:

```
                                       ---------------
                                       |      a      |
                                       |-------------|
                                       |      b      |
                                       |-------------|
                    closure #0 ------> | code for #0 |
                                 |     ---------------
               ---------------   |
closure #2 --> | code for #2 |   |
               |--------------   |
               |      d      |   |
               |-------------|   |
               |      c      |   |
               |-------------|   |
               |    link     |----
               |-------------|
closure #1 --> | code for #1 |
               ---------------
```

where memory grows from the bottom of the figure towards the top.

Putting the two code pointers in the same closure allows them to share the environment and saves three words in the heap. The drawback to this layout is that an assumption is being made that closures #1 and #2 have similar lifetimes. If that assumption proved to be incorrect – if one procedure became inaccessible before the other – at least one word of heap storage would be wasted as long as the other was retained. In actual code, the assumption is true much more often than not.

4.1.2 Global Environment

Recall that one of the advantages of *assignment conversion* was that the value of a variable could exist in any number of closures or registers without regard to assignments to the variable, but that assignment conversion was not performed on global variables. We can divide global variables into two classes: *mutable* and *immutable*. The variable is considered immutable if its value can only change due to interactive debugging. Such a variable can be treated almost like a bound variable; it can be assigned to a register or to a slot in a stack-allocated closure. If the variable is assigned during debugging, the result of the change will be seen eventually.

The situation is different if a global variable is actually assigned as part of the execution of a program. In that case, an assignment to the variable must be effected before the next reference, by updating all machine locations that contain the value of the variable, and this information must be available at run-time. This is not feasible if the value of the variable can be scattered through registers and closures, so these global variables are only allocated to the global environments created when modules are installed in the run-time system.

The result is that an assignment to a mutable global variable takes time proportional to the number of installed modules that reference it. The benefit of this strategy is that *all* references to global variables need not be indirect through a cell. If it is important to have fast assignment to a global variable, the programmer can provide a declaration that will cause a cell to be introduced for that variable.

4.2 Escape Analysis

Escape analysis is the method for determining whether a closure may be stack or register-allocated, or must be heap-allocated. A procedure is said to *escape*, with respect to the compilation, if the compiler cannot identify all the places where the procedure is being called. If all calls are not known, the procedure is said to escape *downward only* if it is known that the current continuation will not be invoked before the procedure becomes inaccessible. This condition is exactly the condition placed on procedural arguments in Pascal by the language definition.

A procedure in call position is a *known procedure* if either:

- It is a lambda-node.
- It is a reference to a variable that the compiler can prove will be bound to a particular lambda-node.
- It is bound to a variable for which there is early-binding information.

Otherwise, it is referred to as an *unknown procedure*.

Escape analysis is done on variables and lambda-nodes. A lambda-node escapes upward if one of the following hold:

- It is an argument to an unknown procedure.
- It is an argument to a known procedure and the corresponding formal parameter escapes upward.
- It is stored into a data structure.
- It is assigned or bound to a variable that escapes upward.

It escapes downward only if none of the previous conditions hold but one of the following does:

- It is an argument to a known procedure and the corresponding formal parameter of that procedure does not escape.
- It is bound or assigned to a variable that escapes downward.

If none of the previous conditions hold, the lambda-node does not escape. A *variable* escapes upward if it is a global variable or any of its references satisfy the same conditions that would indicate that a lambda-node escaped. In addition, a variable escapes if it is referenced in the body of a lambda-node which escapes. Using these rules we can determine whether the arguments to any Scheme procedure escape.

A subtle problem arises with variables introduced by the compiler during CPS conversion to represent continuations. Remembering that a variable will not escape if all places it is being called are known at compile-time, it is safe to say that a continuation passed to a procedure which is open-coded does not escape, because the compiler will be generating the *only* call to the continuation after emitting the code for the procedure body. In most languages other than Scheme, we could assume that no continuations escape upward, because the programmer has no access to continuations. Continuation variables introduced by the compiler would always be either called or passed as a continuation argument to another procedure, and thus the assumption that continuations escape downward is consistent. Unfortunately (for the implementor) Scheme provides the procedure **call-with-current-continuation**, which allows a continuation to be bound to a variable which may escape upward. The upward continuation problem and its solutions will be discussed in a later section. For now, we will assume that the presence of **call-with-current-continuation** does not prevent continuations from being treated in a stack-like manner.

The three possibilites for allocating closures have been discussed and ordered with respect to space efficiency, as well as cost of allocation and deallocation. As we will see, the situation is not as simple as determining the most efficient strategy according to these criteria, because the increase or decrease in *code size* and *number of memory references* in the code must also be taken into account.

4.3 Preliminary Assignment of Strategies

As a first step, each lambda-node in the node tree is assigned a strategy from the set $\{$*heap, stack, registers*$\}$. This initial assignment depends only on the lifetime of the procedure, i.e. whether or not it escapes. Allocating the environment in the registers implies that a non-standard calling sequence may be used when the procedure is called, which is only possible if the locations of all calls are known, i.e. the procedure does not escape. If it does escape, but only downward, the procedure can be stack-allocated. Otherwise, the procedure is of unlimited extent and must be allocated in the heap.

As an example of a closure allocated to registers, consider the following code which returns the longest tail of a list whose head is **eq?** to **obj**:

```
(define (memq obj list)
  (letrec ((memq
             (lambda (list)
               (cond ((null-list? list) nil)
                     ((eq? obj (car list)) list)
                     (else (memq (cdr list)))))))
    (memq list)))
```

Assume that **null-list?**, **car**, **cdr**, and **eq?** are open-coded. Since all references to the bound variable **memq** are calls, the free variable **obj** may be allocated to a register. When the recursive call to **memq** is compiled, **obj** must be moved into that register before jumping to the code for **memq**. The register allocator can ensure that the move is unnecessary. The result is that **obj** will always be in the same register.

This simple tail-recursion is obviously a loop by another name, and a conventional optimizing compiler would surely achieve the same result. But being able to allocate a free variable to a register is not limited to tail-recursion. Consider the following more general recursion which destructively deletes all objects in a list which are **eq?** to **obj**:

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

If **obj** is allocated to a register on entry to **delq!**, the register allocator can make sure that it is never moved. As in the **memq** example, it is never the case that a free variable in a register *must* be moved.

While analyzing the node tree according to the previous criteria tells us the allowable strategy that is most space-efficient, it may be best to use a more general strategy than is required. Here is the **memq** example changed so that the programmer provides a predicate to use instead of **eq?**:

```
(define (mem pred obj list)
  (letrec ((mem
             (lambda (list)
               (cond ((null-list? list) nil)
                     ((pred obj (car list)) list)
                     (else (mem (cdr list)))))))
    (mem list)))
```

It would be possible to keep the free variables **pred** and **obj** in registers because all references to **mem** are calls, but each time **pred** was called, **pred** and **obj** would have to be moved out of their registers because their values are needed by the continuation to the call. When **mem** was called again, **pred** and **obj** would have to be restored to their registers as part of the calling sequence. We would have the benefit of referencing the values of **pred** and **obj** in registers instead of memory, but the cost of pushing the values into memory and then restoring them is incurred *each iteration*. Instead, the closure for **mem** could be allocated on the stack by pushing the values *before* the first call to **mem**. It is a funny kind of closure because although stack space is needed for the values of **pred** and **obj**, a code pointer is not necessary because the address of the code for **mem** is known. When the continuation to the **letrec** is invoked, these values will be popped off the stack. This example motivates several different kinds of stack-allocated closures, each having different behavior with respect to code generation, that will be defined in the next section.

4.4 Strategy Stack

A stack-allocated closure is classified according to whether or not a code pointer is needed and how variables in the closure are referenced. Four different strategies result from this classification:

1. There is a code pointer in the closure and variable references are through the stack pointer (SP). This case will be referred to as *stack/continuation* and is only used for continuations. The continuation must be called using the standard calling sequence for continuations.

2. There is a code pointer in the closure and variable references are through the environment register (P). This case will be referred to as *stack/downward* and is used for procedures that are not continuations and escape downward only. The closure must be called with the standard calling sequence.

3. The closure has no code pointer and variable references are through SP. This case will be referred to as *stack/loop* and is used for procedures that are "loops", or "tail recursions".

4. The closure has no code pointer and variable references are through a frame pointer that could be any register. This case will be referred to as *stack/recursion* and is used for general recursions.

The last two cases are used only when the closure does not escape, so any calling sequence can be used. It is important to notice that being able to use the stack-pointer to reference closure variables is equivalent to the assertion that the closure is always invoked with the same continuation, because the register pointing to the continuation *is* SP.

4.4.1 Strategy/continuation

We have seen that a continuation must be stack-allocated if it, or a variable it is bound to, is passed to a procedure that is not open-coded. There is another case where stack-allocation is better than registers even though it is not required. Consider the following fragment:

```
(lambda (x y)
  (let ((z (if (baz x) x y)))
    (bar (foo x) y z)))
```

After CPS conversion this becomes:

```
(lambda (k x y)
  (let ((join-point (lambda (z)
                      (foo (lambda (v) (bar k v y z))
                           x))))
    (baz (lambda (t)
           (if (true? t)
               (join-point x)
               (join-point y)))
         x)))
```

Assume that **true?** is open coded. Because all of the references to **join-point** are calls, the environment for the closure it is bound to (the values of **x** and **y**) could be kept in the registers. That would mean moving these values into registers when **join-point** is called. However, since the first thing that **join-point** must do is save **y** and **z** because of the call to **foo**, allocating **x** and **y** to registers would be pointless. It would be much better to assign *stack/continuation* to the lambda-node **join-point** is bound to. Then **x** and **y** would be saved on the stack before **baz** is called. This observation leads to the following heuristic:

A lambda-node is assigned stack/continuation *if it, or a variable it is bound to, appears as a continuation to a non-open-coded procedure, or the execution of* *the body of the lambda-node will cause the registers to be saved.*

Variable references in a closure with *stack/continuation* are easy because no explicit environment pointer is needed. The stack-pointer always points to the environment. If a free variable needs to be fetched from a superior environment that is also stack-allocated, a single instruction will do because nested continuations are allocated on the stack contiguously.

4.4.2 Strategy/downward

In general *stack/downward* can be assigned to closures that do not escape upward, but we have to be careful to remain properly tail-recursive. Consider the example:

```
(define (foo x list)
  (map (lambda (z) (bar x z)) list))
```

After CPS conversion:

```
(define (foo k x list)
  (map k
       (lambda (k' z) (bar k' x z))
       list))
```

If we knew that the second argument to **map** (after CPS conversion) did not escape upward it could be assigned *stack/downward*. The closure would be pushed on the stack before the call to **map**, but when would the stack space be reclaimed? To remain properly tail-recursive, before calling **map** the stack-pointer must be restored to the value it had when **foo** started executing. Since the argument to **map** must exist on the stack while **map** is executing, we have a contradiction. This type of closure must be heap-allocated. On the other hand, if the code were:

```
(define (foo x list)
  (baz (map (lambda (z) (bar x z)) list)))
```

After CPS conversion:

```
(define (foo k x list)
  (map (lambda (v) (baz k v)
       (lambda (k' z) (bar k' x z))
       list)))
```

the argument to **map** could be allocated on the stack as a closure internal to the continuation to **map**.

In closures with *stack/downward*, variables are referenced in the same way as in those with *stack/continuation* because of the same easy reference to superior stack-allocated closures. The difference is that in the *stack/downward* case, variable references must go through the environment register because the calling sequence for a procedure is different from that of a continuation.

4.5 Loops and Recursions

In the context of Scheme, what is a loop? In other languages, including LISP, specific iteration constructs are provided. Compilers spend a lot of time optimizing the code for these constructs. The problem is that these loops are not very general. In fact, these constructs are the only way iterations can be expressed with a guarantee that the stack will not overflow. In Scheme, as far as the compiler is concerned, all iterations are expressed as recursions, even if syntax is provided which looks like the iterative constructs of other languages. The only constructs that Orbit analyzes are sets of mutually recursive procedures.

4.5.1 Loops

Some sets of mutually recursive procedures have the useful property:

Definition 4.1 *A set of mutually recursive procedures that are only called, and always with the same continuation, is called a* loop.

Because the stack-pointer always holds the continuation, it has the same value each time one of the loop's procedures is called. This is precisely the condition needed to assign a procedure *stack/loop*. It allows the free variables of the loop to be pushed on the stack before entering the loop and to be referenced through the stack-pointer. When one of the loop procedures finally invokes its continuation or passes it out of the loop, code is emitted to pop off the free variables.

Note that any set of mutually recursive procedures satisfying the definition of a loop could also be given strategy *registers*. The actual strategy used depends upon whether or not the free variables will need to be saved and restored during execution of the loop. The **mem** example looked at previously would be given *stack/loop*, but if we change that example slightly to:

```
(define (memq obj list)
  (letrec ((memq
             (lambda (list)
               (cond ((null-list? list) nil)
                     ((eq? obj (car list)) list)
                     (else (memq (cdr list)))))))
    (memq list)))
```

where **eq?** is open-coded, **obj** could be kept in a register throughout the execution of the loop, never being moved. Thus **memq** could be given strategy *registers.*

4.5.2 General Recursions

What happens if we relax the condition that requires each mutually recursive procedure to be called with the same continuation? We are now considering code such as:

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

Two calls to **del!** pass the original continuation but one passes a continuation that does a **set!** and invokes its continuation with argument **list**. It is still possible to push the values of the free variables, **pred** and **obj**, before entering the recursion, but they cannot be referenced through the stack-pointer because it will be at a different place each time **del!** is called. The values must be referenced through a frame pointer that needs to be saved when **pred** is called. It is also no longer possible to know at compile-time where code to pop off the environment should be emitted because it is impossible to tell when the recursion is “really ending”; i.e., we do not know whether the continuation being invoked is the original one passed in to **del!** or one of the others.

One solution would be to give the stack-allocated environment a code pointer that would pop the closure off the stack and return. In the **del!** example this would work, but if we changed the code to:

```
(define (del! pred obj list)
  (letrec ((del!
             (lambda (list)
               (cond ((null-list? list) (foo '()))
                     ((pred obj (car list))
                      (del! (cdr list)))
                     (else (set! (cdr list)
                                 (del! (cdr list)))
                           (foo list))))))
    (del! list)))
```

the environment would not be popped off until **foo** or some other procedure eventually invoked this "extra" closure.

The solution we have chosen is to push a special marker onto the stack after pushing the environment [Rozas]. The code to return from **del!** checks to see if the marker is at the top of the stack and, if so, it pops off the environment before invoking the "real" continuation, i.e. the one that was passed to **del!**. This problem is really the same as the one that occurred with *stack/downward*, but in that case there was no solution because we could not know that the procedure being called would generate the runtime checks before invoking its continuation.

The properties of the various closure strategies are summarized in the following table:

Strategy	Closure register	Calling convention	Runtime check on return?
Registers	none	any	no
stack/Continuation	SP	standard	no
stack/Loop	SP	any	no
stack/Recursion	FP	any	yes
stack/Downward	P	standard	no
Heap	P	standard	no

The closure register column indicates how free variable references are compiled, i.e. which register points to the closure when it starts executing. The use of FP in *stack/recursion* is very similar to the use of a frame pointer in a conventional compiler, but in Orbit it can be any register and a frame pointer exists only while in a closure with this particular strategy.

4.6 Final Assignment of Strategies

Consider the example:

```
(define (f x y)
  (letrec ((foo
             (lambda (z)
               (cond ((null? z) nil)
                     ((bar x y)
                      (baz (lambda (w)
                             (if w x (foo w)))))
                     (else (foo (cdr z)))))))
    (foo y)))
```

Even though all references to **foo** are calls, it cannot be assigned *stack/recursion* (stack-allocated) because it is called from inside the procedural argument to **baz**, which escapes upward. This example demonstrates that the strategy assigned to a lambda-node bound to a variable depends on the type of closure the variable might be called from.

4.6.1 Strategy Analysis

We place an ordering on the closure strategies for local recursions as follows, where $\prec$ indicates less general:

$$stack/loop \prec stack/recursion \prec heap$$

The strategy assigned to a procedure must be at least as general as any from which it is called, unless it is strategy *registers*; it is always the case that any procedure that is only called *may* be assigned strategy *registers*. This ordering implies that the strategy initially assigned to a lambda-node must be modified based on the strategy of each lambda-node whose body it is called from. A description of the restrictions imposed by being called from each strategy, and the strategy changes thus induced, are as follows:

strategy registers. Being called from a closure with strategy *registers* imposes no restriction if the closure was not bound in a **letrec** or is a *loop*. If the closure represents a recursion, then any procedure called from it cannot be given *stack/loop* because the stack-pointer might be different for different calls to that procedure.

stack/continuation and stack/loop. These cases use stack-allocated environments. If a procedure with strategy *registers* were called from here it would mean that the registers containing the environment would have to be saved and restored. The strategy of such a procedure would thus be changed to *stack/loop* if it were a *loop*, otherwise it would be changed to *stack/recursion*.

stack/recursion. While executing a procedure with this strategy, the stack-pointer could be anywhere. If it contains a call to a procedure with *stack/loop*, that procedure must be changed to *stack/recursion* because *stack/loop* assumes that the stack-pointer is always the same when the procedure is invoked.

stack/downward. If we had a call to a procedure with *stack/loop* here the call would have to move the stack-pointer past this closure. Therefore that called procedure must be changed to *stack/recursion*.

strategy heap. As in the previous example, a procedure called from here needs to be changed to strategy *heap* unless it is strategy *registers*.

The algorithm for strategy analysis uses these rules by examining all calls to procedures that are bound to variables, making changes to the assigned strategies, and then iterating until no more changes in strategy are made.

However, the rules given are not quite adequate. To see why we have to look at the question of live variables.

4.6.2 Live-variable Analysis

In order to do closure analysis we must determine which variables are live for each lambda-node in the node tree. As discussed previously, a pointer to a closure is like a variable, in the sense that it needs to be accessible whenever one of the variables in its environment needs to be accessible. For live-variable analysis several functions are needed:

closure-variable takes a lambda-node as argument and returns the variable that represents the closure corresponding to that node.

variable-known-lambda takes a variable as argument and returns the lambda-node to which it is known to be bound, if any. Otherwise it returns false.

lambda-live takes a lambda-node and returns the set of variables previously computed to be live at that lambda-node. This distinction is important if the algorithm is performed iteratively.

reference-variable takes a reference-node and returns the variable it is a reference to.

set-of takes some number of values and returns a set containing those values.

The value **global** indicates that a pointer to the global environment is required. The following algorithm will find the set of live variables at each lambda-node:

```
(define (live-analyze node)
  (case (node-type node)
    ((lambda)
     (let ((live (live-analyze (lambda-body node))))
       (if (empty-set? live)
           global
           (set-difference live (formal-parameters node)))))
    ((call)
     (map-union live-analyze (call-proc+arguments node)))
    ((leaf)
     (live-analyze-leaf node))))
```

```
(define (live-analyze-leaf node)
  (case (type-of-leaf node)
    ((primop)
     empty-set)
    ((literal)
     (if (addressable? (literal-value node))
         empty-set
         global))
((reference)
 (let* ((variable (reference-variable node))
        (lambda (variable-known-lambda variable)))
   (if (not lambda)
       (cond ((and (global-variable? variable)
                   (mutable? variable))
              (set-of variable global))
             ((continuation-variable? variable)
              empty-set)
             (else
              (set-of variable)))
       (case (lambda-strategy lambda)
         ((registers)
          (lambda-live lambda))
         ((loop continuation)
          empty-set)
         ((recursion)
          (set-of (closure-variable lambda)))
         ((heap)
          (if (all-are-global? (lambda-live lambda))
              global
              (set-of (closure-variable lambda)))))))))
```

In the last **case**, the empty set is returned for *stack/loop* and *stack/continuation* because we can always use the stack-pointer to reference the closure. Similarly, we can ignore a variable that is bound to a continuation because fetching the variable involves, at most, adjusting the stack-pointer. Recall that every closure in an expression being compiled may have two environments where values are obtained: the local environment, with variables that are bound within the expression, and the "global" environment which contains:

- Variables not bound within the expression.
- Literal values (constants and quoted structure) that cannot be referenced as an immediate operand in the instruction stream.
- Closures which are to be heap-allocated and have no free variables bound within the expression.

Whenever the algorithm encounters a value that will be found in the global environment and cannot be assumed constant, it just returns the value *global.* These variables cannot be assigned to registers or closure slots because the result of a side-effect would not be "seen".

It is important to note the way a variable, known to be bound to a lambda, is treated in the analysis, i.e. that variable never occurs in a set of live variables. Instead, the variable is *identified* with the lambda to which it is bound. If that lambda will have its environment in registers, the values in the environment must be accessible at the point of the call. Hence the live variables associated with the variable reference are those needed by the lambda to which it is bound.

If the environment is not in registers, all that is needed to reference the variable is a reference to the lambda to which it is bound. Recall that each lambda-node has a formal parameter that refers to itself; environment pointers are treated like variables. If the lambda has *stack/loop* or *stack/continuation*, nothing is needed to call it because the environment will be accessed through the stack pointer. Otherwise, the variable representing the lambda must be considered live. As an example of what this means, consider the code:

```
(define (foo x)
  (letrec ((bar (lambda (y)
                  (if (not y) x bar))))
    bar))
```

Most compilers would take the view that the free variables of the lambda were **x** and **bar**. Both of these values would be stored in the closure. The resulting code might look like:

```
    COMPARE-TO-FALSE  A1           ;;; (not y)
    JUMP-ON-EQUAL     L1
    MOVE              X(P),A1      ;;; return x
    RETURN
L1: MOVE              BAR(P),A1    ;;; return bar
    RETURN
```

But if we view the code as being:

```
(define (foo x)
  (letrec ((bar (lambda (bar y)
                  (if (not y) x bar))))
    bar))
```

and use the algorithm presented, there is only one free variable, **x**, and we get the following code:

```
    COMPARE-TO-FALSE  A1          ;;; (not y)
    JUMP-ON-EQUAL     L1
    MOVE              X(P),A1     ;;; return x
    RETURN
L1: MOVE              P,A1    ;;; return bar
    RETURN
```

This example clearly shows how environment pointers are treated just like other variables; the value being returned is just the value in the environment register (P)!

This algorithm must iterate because procedures can be bound to variables. The fact that it starts out with the assumption that the set of live variables at each lambda node is empty implies that the algorithm must iterate until convergence. It is desirable that the live-variable information be exact, i.e. no variable is indicated as live where it is not. Minimal live variable sets are not necessary for correctness, but they are necessary to achieve the goals of reducing the size of heap-allocated closures and keeping as many variables in registers as possible.

Clearly the live-variable analysis depends on what strategies have been assigned to lambda-nodes. Unfortunately, the converse is also true. To see why we look at the **del!** example given in the last section again, but change it slightly:

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

The change is that **obj** is no longer free to **del!**. In the original example the best strategy was *stack/recursion*, but if we have only one free variable that would mean saving a pointer to a single variable across the call to **pred**. Obviously it would be better to just save **pred**, i.e. use strategy *registers*. In this example, as in the others, it is sufficient to do a static analysis of the code to arrive at the best strategy. Unfortunately, this is not always the case. In the section with the example:

```
(define (f x y)
  (letrec ((foo
            (lambda (z)
              (cond ((null? z) nil)
                    ((bar x y)
                     (baz (lambda (w)
                            (if w x (foo w)))))
                    (else (foo (cdr z)))))))
    (foo y)))
```

it was observed that **foo** could not be assigned *stack/recursion*, but what strategy should it be assigned? The two possibilities are strategy *registers* and strategy *heap*. If we use strategy *heap* the closure structure for **foo** and the argument to **baz** would be:

```
                                  --------
                                  |  x   |
                                  |------|
                                  |  y   |
                                  |------|
                             ---> | code | <---- foo
                             |    --------
                   --------  |
                   |      |---
                   --------
argument to baz -> | code |
                   --------
```

If we used strategy *registers* it would be:

```
                   --------
                   |  x   |
                   -------|
                   |  y   |
                   -------|
argument to baz -> | code |
                   --------
```

With strategy *heap* the amount of space used is 3+(# of times baz is called)∗2 and with strategy *registers* it is (# of times baz is called)∗3. Which is better depends on the dynamic behavior of the function **f**. This sort of decision, of course, arises often in the compilation of any language. In Orbit we assume that because **foo** is a recursive procedure, the closure for the argument to **baz** will be created much more frequently than the closure for **foo**.

Obviously, the rules for strategy analysis have to be modified to take the live variable information into account. In order to get everything right the strategy analysis and live-variable analysis *have to be done at the same time.* Therefore the two algorithms are combined. If we are not careful, the combined algorithm might not terminate because changing the strategy of a procedure to strategy *registers* may increase the number of live variables at the lambda-nodes containing calls to it. That increase in live variables might cause a procedure to be changed from strategy *registers* to one of the others. This change might cause the original procedure to flip back to what it was, and so on. The solution is to disallow a transition to strategy *registers* from any other.

4.7 Closure Hoisting

Thus far, the issue of *when* a heap or stack-allocated closure will be allocated has not been addressed. The most straightforward approach would be to allocate the storage when the lambda-node to which it evaluates is encountered in the node tree. Two examples will show why a more complicated analysis is needed.

```
(letrec ((fact
           (lambda (n)
             (if (= n 0) 1 (* n (fact (- n 1)))))))
  (fact 5))
```

After CPS conversion the procedure fact looks like:

```
(lambda (k n)
  (conditional (k 1)
               (- (lambda (v)
                    (fact (lambda (w)
                            (* k n w))
                          v))
                  n
                  1)
               (= n 0)))
```

Assume that $+$, -, and $=$ are open coded and that **fact** has been assigned strategy *registers*. The continuation to - will be strategy *registers*, but the continuation to the call will have to be *stack/continuation*. Assuming that the argument **n** of **fac** is allocated to the first argument register, **A1**, and remembering that the first argument to a continuation goes in the first argument register, the code for **fact** might look like:

```
FACT:
    COMPARE-TO-ZERO A1         ;;; (= n 0)
    JUMP-ON-EQUAL   RETURN
    MOVE            A1,R2      ;;; (- n 1)
    SUBTRACT        1,R2
    PUSH            A1         ;;; save n and push code
    PUSH-ADDRESS    CONT       ;;; descriptor for continuation
    MOVE            R2,A1       ;;; call fact, n is in R1
    JUMP            FACT

CONT:
    MULTIPLY        4(SP),A1   ;;; return (* n w)
    POP-STACK       8          ;;; remove continuation
    RET                        ;;; return

RETURN:
    MOVE            1,A1       ;;; return 1
    RET
```

The code at the label **FACT** has two extra move instructions in it. We could not just subtract 1 from R1 because the value in R1 (n) was still live *but only because it was needed by the continuation.* If the evaluation of the continuation were moved up to the point before the subtraction, the following code would result:

```
FACT:
    COMPARE-TO-ZERO A1        ;;; (= n 0)
    JUMP-ON-EQUAL   RETURN
    PUSH            A1        ;;; save n and push code
    PUSH-ADDRESS    CONT      ;;; descriptor for continuation
    SUBTRACT        1,R2      ;;; (- n 1)
    JUMP            FACT
```

This example is illustrative of the fact that a continuation should be allocated as soon as it is known that it will have to be allocated and doing so would reduce the number of live variables. We call this *closure hoisting*. For the case of a continuation there would be no point in raising its evaluation above a conditional expression. This is because the properties that make a stack useful for continuations prevent these continuations from sharing closures.

The situation is different with heap-allocated closures. Normally, when we hoist a continuation, we do not have to keep track of a pointer to it because it is contiguous with the current continuation. However, if a closure is heap-allocated, we need to store its pointer somewhere until it is used, making early allocation not as attractive. We also might want to hoist a heap-allocated closure above a conditional expression if it could be merged with another. As an example consider:

```
(lambda (x y)
  (let ((foo (lambda (z)
               (if (eq? z 'x)
                   (lambda () x)         ;;; lambda #1
                   (lambda () y)))))     ;;; lambda #2
    foo))
```

If we evaluated the lambda-expression returned by calling **foo** after testing to see which lambda-expression was being returned we would have to use heap space whatever the result of the test, but if we raised the lambda-expressions above the conditional expression we could have the closure structure look like:

```
----------------
| code for #2  |
|--------------|
| code for #1  |
|--------------|
|      y       |
|--------------|
|      x       |
|--------------|
| code for foo |
|---------------
```

This choice increases the size of **foo** by two words but calling **foo** now causes no additional allocation at all. It simply returns a pointer to the appropriate slot in the bigger closure. As before, we assume that lambdas #1 and #2 will be evaluated many more times than the lambda bound to **foo**.

Thus far we have been concerned with early evaluation of lambdas, but often it is better to try to delay the evaluation as long as possible. As an example, consider the following code:

```
(define (f x)
  (let ((new-x (if (fixnum? x)
                   x
                   (get-a-fixnum))))
    .. x .. new-x ..
    ))
```

After pseudo CPS conversion this becomes:

```
(define (f x)
  (let ((j (lambda (new-x) .. x .. new-x ..)))
    (if (fixnum? x)
        (j x)
        (get-a-fixnum j))))
```

Because **j** appears as the continuation to the call to **get-a-fixnum**, the closure for **j** must be assigned *stack/continuation*. However, the code could be transformed into:

```
(define (f x)
  (let ((j (lambda (new-x) .. x .. new-x ..)))
    (if (fixnum? x)
        (j x)
        (get-a-fixnum (lambda (x) (j x))))))
```

Now all of the references to **j** are calls and so **j** can be assigned strategy *registers*, while the **(lambda (x) (j x))** is given *stack/continuation*. The stack allocation of **j** has been effectively pushed below the conditional, only ocurring if necessary. The compiler performs this sort of transformation whenever possible.

4.8 Representation Analysis

After closure strategy analysis we know where each closure will be allocated. The remaining task is to determine which closures should share an environment, which variables should be in the environment, and how much space they need to occupy. In addition, we need to determine which cells, introduced for bound variables that might be side-effected, can be eliminated.

4.8.1 Representation of Variables

The fact that each value in a program must be represented as a tagged pointer can lead to great inefficiencies in compiled code, because conventional machine architectures do not provide intentional support for manipulating tagged quantities. One can play various tricks with pointer tags to minimize the problem but, on conventional machine architectures, it cannot be avoided. In the T3 system the low order two bits of a pointer are used as the tag field. In particular, these two bits are zeroes for fixnums. Thus most arithmetic and boolean operations can be done directly on the tagged pointers, with overflows detected naturally. There are several important exceptions, however, multiplication of fixnums being a good example. To multiply two tagged fixnums to yield a tagged fixnum, the tag field of one of them must be right-shifted out. Consider iterative factorial as an example:

```
(define (factorial n)
  (letrec ((fac
             (lambda (result i)
               (if (= i 0)
                   result
                   (fac (* result i) (- i 1))))))
    (fac 1 n)))
```

By default, **result** and **i** would be represented as tagged values and the result of the multiplication would have to be shifted left 2 bits. Rather than doing the shift in each iteration, the variable **i** could be represented in unshifted form, eliminating the need for the shift after the multiply. Of course, this could only be done if it were known from type declarations that the result would always be a fixnum.

Representation analysis determines which variables can be handled more efficiently using a non-tagged representation. Non-tagged representations are only a possibility for variables with a known type. The type information is either supplied by the programmer, or is inferred from a simple first-order analysis. The analysis is based on the various primops, with each primop supplying information about what representations it would prefer or require from its operands and which it can or must deliver to its continuation. A character, for example, could be stored as an eight bit ascii value, and several of them could be packed into a single word of a closure. This analysis is especially important in order to do floating-point operations efficiently, because converting a number from the form used by machine instructions to a T pointer requires allocating storage.

4.8.2 Collapsing Cells

The front-end of Orbit introduced cells for all bound variables that might be assigned. In general, each cell has to be allocated in the heap although this might not be necesary. Indeed, it is possible to do arbitrarily large amounts of analysis, as conventional optimizing compilers do, in order to figure out exactly which cells are not really needed. I decided to analyze only the cases that would occur in programs written in the normal "Scheme style". A bound variable could be assigned for one of two reasons:

1. It represents the local state of an object.
2. The programmer decided to give a new value to the variable by reassignment rather than by rebinding.

The code produced for programs that use assignments to do rebinding, which is unfortunately a common practice in traditional Lisp code, will be relatively poor. The other case is illustrated by the example:

```
(define random
  (let ((seed 34343))
    (lambda ()
      (set! seed (make-new-seed seed))
      seed)))
```

A cell will be introduced for the assigned variable **seed**, but because this variable is only referenced in one closure the cell can be collapsed. As a result, the closure contains the value of **seed** instead of a pointer to a cell containing the value. This saves both the heap space for the cell and the time required for indirection when manipulating the value. The following heuristic is used for collapsing cells:

A variable holding an introduced cell may be collapsed if all of the following hold:

1. The value of the variable will only be in one closure.

2. That closure is in the heap.

3. All references to the variable are either to get the contents of the cell or change the contents.

If the third condition fails it means that the cell may be being used by the programmer as a location, and so cannot be collapsed. Because of **call-with-current-continuation**, cells bound to variables in stack-allocated closures may not be collapsed, as is shown in the next section.

4.8.3 Closure Representation Analysis

There are three different layouts for heap-allocated closures at runtime. Which one is used depends on the nature of the live variables of each closure. If there are no live variables or all are global, the closure will be internal to the global environment. Evaluating the lambda-expression will cause a pointer into the global environment to be fetched. If none of the live variables are global, the closure will have an ordinary code descriptor, but if there are some global and some non-global live variables, the code descriptor will also be a pointer into the global environment. This trick is used because many closures reference the global environment in addition to their local environment. This optimization is important because lexical scoping of global variables implies that a procedure referring to global variables *must* hold a pointer to the global environment.

Each closure residing on the stack or in the heap at runtime has a parallel structure in the compiler with the following information:

code-descriptors A list of the lambda-nodes whose code descriptors will share the closure, starting with the one whose code descriptor will be the first slot.

variables A list of all the variables with values in this closure and their respective offsets in the closure.

rootable The number of rootable slots.

non-rootable The number of non-rootable slots.

global offset If variables are being referenced in the global environment then the code descriptor in the first slot will also be a pointer into the global environment as described previously. The global-offset is the offset of this code descriptor in the global environment.

link If the closure will have a pointer to a superior closure, this value is the compiler representation of that superior closure.

Each lambda-node knows of which closure structure it is a part as well as the runtime offset of its code descriptor.

The compiler structure for lambda-nodes with strategy *registers* is simpler, containing:

variables A list of the arguments and free variables and places for the corresponding register assignments. These places will be filled in during code generation.

link If some variables will be accessed from another closure this field contains the structure representing that closure.

Closure representation analysis is performed via a top-down walk over the node tree. Lambda-nodes (except for those with strategy *registers*), evaluating to closures that will be packed together, are collected and the compiler structure is created for that closure. All lambda-nodes that occur as arguments to the same call node are assigned to the same closure. This includes grouping together lambda-nodes bound in a **let** or **letrec**, but only those that escape need to have code descriptors in the closure.

When a closure is to be created, we need to decide whether a value that is needed by the closure, but which has already been stored in a superior closure, will be copied or referenced through a pointer to the superior closure. Because of assignment conversion, any variable may be copied without worrying about possible assignments. As we walk down the tree, each lambda-node is presented with two sets of variables:

1. Those that have already been placed in a superior stack-allocated closure.

2. Those that have already been placed in a superior heap-allocated closure.

These sets must be maintained separately because a variable live at a heap-allocated closure must be copied into the heap if it has been stored on the stack.

The costs of copying (move instructions and extra storage requirements) must be balanced against the cost of leaving a pointer (more expensive variable references). Heap storage is so expensive that variables are never copied from one heap closure to another, unless there is only one. For closures with *stack/loop* and *stack/recursion*, only stack space is used. Furthermore, the closure has to be filled with values only once, though it will presumably be called many times. For these reasons no pointers are kept in such closures, except possibly for a pointer to the global environment. Such a pointer must always be maintained at a point where a global variable might be modified. The same reasoning applies if the environment will be kept in registers.

For the other cases, namely *stack/continuation* and *stack/downward*, it is not possible to be sure what will be best. A form of *strictness analysis* [Mycroft 81] could be done, and variables that were certain to be referenced could be copied, but I judged, for just this case, that analysis not to be worth doing.

4.9 Call-with-current-continuation

The Scheme procedure **call-with-current-continuation** is a very powerful construct that can be very expensive to implement. Because this language feature allows continuations to be manipulated by the programmer in arbitrary ways, continuations cannot, in general, be treated in a stack-like manner. The most straightforward approach is to actually heap-allocate continuations. This strategy is prohibitively expensive in both time and space, *even if no calls to* call-with-current-continuation *are ever made.* An alternative is to "freelist" continuations (stack frames). This improves the heap storage problem but still must always be paid for.

The other alternative, the one used here, is to have a single stack. When a continuation is captured by a call to **call-with-current-continuation**, the active portion of the stack is copied into the heap. Each time the continuation is invoked, the stack which was copied into the heap is copied back into the stack. The advantage of this strategy is that the presence of this language feature incurs almost no cost is, unless it is actually used. The only restriction imposed is that once an object is allocated on the stack, it must not be updated to ensure that all stack references are consistant whenever a continuation is invoked. In particular, this means that cells corresponding to variables in stack closures may not be collapsed. The disadvantage of the stack copying strategy is that the copying and copying back of stacks can be very expensive if they are large.

Chapter 5

Register Allocation and Code Generation

Over the years a systematic method of code generation has been developed for programming languages that allow recursion. Aspects of this method include:

- Parameters to procedures are passed on a stack.
- Register allocation is a separate pass that occurs before the actual generation of code.
- If closures are allowed, the method of referencing the variables in the closure is determined during register allocation (by environment chaining) or is fixed (by using a display).
- The *procedure call* is a fundamental concept. Procedures return a single value in a register or on the stack.

There is reason to believe that some of these aspects might have actually impeded the development of programming languages, e.g., the lack of support for procedures returning more than one value and for "first-class" procedures. Implementation of these features in the "conventional" model would be inefficient or difficult, or both. The widely held view that procedure calls are expensive has certainly influenced the way programs are written. In this chapter I will argue that all of the above aspects of the conventional model are either unnecessarily restrictive or inherently inefficient.

5.1 Fundamental Choices

This section will examine some of the fundamental design decisions that have to be made regarding code generation. These decisions cannot be made independently because of their strong interaction.

5.1.1 Where Should Parameters be Passed?

Traditionally, arguments to procedure calls have been passed on a stack. In the early days of computing, machines did not have general purpose registers, and passing arguments on a stack fits naturally into the implementation of recursion using a stack. Even though modern, optimizing compilers put great effort into register allocation in order to use the higher-speed registers as much as possible, arguments are still generally passed on the stack. Two important exceptions are the MACLISP and PSL [Griss 82] compilers which used registers for argument passing. These two compilers were also noted for the efficiency of the code they generated.

The two approaches take different views of the machine architecture. The compiler that uses the stack for argument passing treats the stack as the fundamental storage for variable values. Values are cached in registers when possible because register operations are faster. On the other hand, a register-based compiler assumes computation is done in registers. Only when a context-switch (unknown procedure call) occurs is it necessary to force values out of registers onto the stack.

Passing arguments on a stack simplifies the compiler because arguments are located using a frame pointer, but may force arguments to be in memory when not necessary. Passing arguments in registers requires more work from a register allocator, but the code generator now has an opportunity to place the results of computations in the the registers where they will be passed as arguments to other procedures.

On the other hand, it is possible to construct code that would run more efficiently if arguments were passed on the stack. For example:

```
(define (f x y z)
  (g)
  (bar x y z))
```

In this example, if arguments are passed on the stack the code for the body might look like:

```
JSR    G
JMP    BAR
```

but if arguments were passed in registers, it would look like:

```
PUSH   R1
PUSH   R2
PUSH   R3
JSR    G
POP    R3
POP    R2
POP    R1
JMP    BAR
```

Fortunately, a simple empirical analysis will show that passing arguments in registers should significantly increase performance for "typical" code.

Passing arguments on the stack has two clear disadvantages compared to using registers:

- A referenced argument will always have to be fetched from memory at least once.
- In a tail-recursive call, even though some arguments might not need to be pushed onto the stack, any argument manipulation will be in memory instead of in registers.

For the purposes of the analysis, these two disadvantages will be ignored; tail-recursive calls are assumed to be optimized even if arguments are passed on the stack. What will be measured is how often values must be pushed onto the stack.

For the analysis it is also necessary to make a distinction between user variables and "temporaries", i.e. variables that are bound by continuations. This distinction is necessary because the "temporary" values are treated the same, regardless of the argument passing strategy used. These values are always generated in registers and, if needed after an unknown procedure call, will have to be pushed on the stack in either case.

As it generates code, the compiler keeps track of:

1. The number of times any value appears as an argument to a non tail-recursive procedure call.
2. The number of formal parameters that are live across a procedure call. Parameters of continuations are not counted because they are the "temporary" variables.

In the first case, if arguments are passed in registers there is no memory reference cost for an argument. Although a move from memory might be needed in this case, the stack case will require the same move, but onto the stack. So, for the stack case, one push is charged for each such argument. Actually it is only fair to charge if, in the register case, the argument is passed in a machine register. Thus, for each procedure call, the charge is the minimum of the number of arguments and the number of machine registers used for argument passing. In practice this doesn't make much difference because most procedures have a small number of arguments. In the stack case, formal parameters to procedures never need to be moved onto the stack, so for the register case one push is charged for each parameter which needs to be saved.

Several modules in the compiler were used as "average code" for the experiment. The following table shows the results of compiling four modules:

arguments pushed (n)	arguments saved (m)	m/n
210	37	.18
323	33	.10
136	25	.18
89	14	.16

Of course these are measurements based on static analysis of the code, but there is little reason to believe that the dynamic behavior will be, on average, any different.

To interpret the data we use the MC68000 processor as an example. In the register case, saving a formal parameter means pushing the contents of a register onto the stack, which costs 16 cycles. To evaluate the cost of pushing an argument there are three relevant cases:

1. The argument is already in the register that it needs to be passed in. There is no cost in such a case.

2. The argument is in a register but must be moved to another. The cost is a register to register move: 4 cycles.

3. The argument is in memory. The cost is a memory to register move: 16 cycles.

In the case of passing arguments on the stack, there is no cost to save a formal parameter because the value in question is already on the stack. When arguments are being passed on the stack, the first case is generally not possible and the cost of the second is that of pushing the contents of a register onto the stack: 16 cycles. The third case is now the cost of a push from memory: 26 cycles. These costs are summarized in the following table:

	arguments in registers	arguments on stack
argument in right register	0	not applicable
argument in wrong register	4	16
argument in memory	16	26

Let **n** be the number of arguments passed and **m** be the number of formal parameters saved. If **rcost** is the cost of a register argument and **scost** is the cost of a stack argument, we have the following formula:

```
cost of registers = n * rcost + 16 * m

cost of stack = n * scost
```

If we assume that each of the three argument cases is equally likely, one computes an average **rcost** of about 7 and an average **scost** of about 19 cycles. Using these values in the above formulae gives the result that the costs are equal when the number of values saved (m) is equal to three fourths the number of arguments passed (n). In other words, it is better to use registers whenever $m/n < .75$. Referring back to the table of results, one can see that for the code profiled, this condition was always true by quite a margin.

Although the 68000 was used as an example, the relevant machine parameter is obviously the relative cost of moves from and to memory, compared to references to values in registers. The static analysis indicates that passing arguments in registers should lead to faster running code than passing arguments on the stack. Benchmarks to support this data will be presented in the final chapter.

5.1.2 When Should Registers be Allocated?

Two register allocation strategies that have become popular in compilers for Lisp as well as other languages are graph coloring [Chaiten 81] [Chaiten 82] [Chow 84] and TN binding [Wulf 75]. In both of these methods, register allocation is viewed as the process of assigning a register or memory location to each user-defined variable or compiler-generated temporary variable. The goal is to make these assignments in such a way as to minimize the number of move instructions, usually by trying to ensure that a value is computed in a location where it can be used in a subsequent operation. Memory references are reduced by allocating frequently used variables to machine registers. After assigning a single machine location to each variable the process of code generation is simple, but the resulting code can have glaring inefficiencies. Consider this simple example of C code:

```
f(x)
int x;
 g();
  return (x+x);
```

In almost all existing C compilers, **x** will be on the stack after the call to **g** returns. A typical sequence would be:

```
jsr g                ;;; call g
move x(sp),r0        ;;; move x into return register
add  x(sp),r0        ;;; why not add r0,r0 ?
pop  x               ;;; pop off the value of x
ret                  ;;; return from f
```

This example shows the main flaw in TN binding or graph coloring: a variable is assigned to exactly one register or exactly one memory location. In other words, each variable is assigned a *single home*. The code generator knows that **x** is on the stack. After the move instruction it still knows that **x** is on the stack, but it doesn't know that it is now in a register also.

In a language like C this is perhaps not too troublesome, since the worst thing that can happen is a reference is made to memory when it could have been to a register. However, the situation becomes much more severe if closures are allowed. In that case *each reference* to a variable in a closure might require several move instructions. This problem could be avoided by using a *display* to reference variables in closures, but that strategy has problems to be discussed in the next section. The reason for the popularity of these register allocation strategies is that they can generate very good code in the absence of closures.

Also, it is clear that in order to do *global* register allocation, some registers must be allocated before code is actually generated.

An alternative is to do code generation and register allocation at the same time; "on-the-fly register allocation". This is not a new idea, having been discussed elsewhere [Aho 86]. It was also discussed from a theoretical point of view [Karr 84] but apparently never implemented. As described in the next section, on-the-fly register allocation can be adapted to deal well with closures. To date, no one has shown how the other register allocation strategies can be extended to languages with first-class procedures in an efficient manner.

Together with appropriate techniques for dealing with closures, the single-home problem can be avoided by doing register allocation and code generation at the same time. The problem is that a global view of register allocation is lost. This problem is ameliorated somewhat by the use of a form of *trace scheduling* to be discussed.

5.1.3 Closures

As far as register allocation is concerned, the difference between a language like Scheme and one like C is the presence of variables that must be fetched from dynamically-created lexical environments. Two methods have been used to generate references to variables that do not reside in registers or stack-allocated temporary locations:

The use of a display. A *display* is the traditional method for handling closures in Algol-like languages. Each procedure keeps a set of pointers to each enclosing environment which has variables that might need to be referenced. This strategy makes register allocation and code generation for these variables easy and inexpensive, but there are some severe problems:

- The display has to be updated each time a new lexical scope is entered.
- If the display is kept in memory, updating it is more expensive, as is referencing a variable in the display. If it is kept in machine regis-

ters, these registers are not available for general allocation, effectively reducing the number of registers *whether or not the display is ever used.*

- If it cannot be determined at compile-time whether an environment variable will be referenced, the display register corresponding to that environment must be loaded even though it may turn out not to have been necessary.

Environment chaining. If a variable may be referenced from an inferior closure, the inferior closure maintains a link to the superior closure containing that variable. Fetching a value involves traversing these links, bringing each link into a register, until the value is accessable. The advantage of this strategy is that no registers are reserved for the special purpose of accessing values in environments, i.e. the potential presence of closures adds no cost unless the closures are used. The disadvantage is that, as we saw previously, if conventional register allocation is used several move instructions may be generated for each variables reference.

Orbit obtains the benefits of both of these strategies by using a *lazy display* strategy, combined with on-the-fly register allocation. The lazy display strategy is a combination of a display and environment chaining. Environment links are kept but are used to load display registers. The two important facts are:

1. A display register is not loaded with an environment pointer *until a reference is made to a variable in that environment.*

2. The number of registers allocated to the display may vary dynamically (at runtime).

In other words, the environment pointers in the display compete for registers with other values, i.e. they are treated like all other variables. The first time a closure variable is referenced, environment links are followed with each environment pointer potentially being assigned to a register. A subsequent reference to the variable, or any other in the same closure, might find the outermost pointer already in a register, obviating the need to bring that pointer into a register. The result is that almost all of the benefits of a display are obtained, but there is no overhead due to the display unless it is used.

5.1.4 Procedure Calls

In the traditional compiler's view, a procedure is a computation that takes some number of arguments— on the stack because of the possibility of recursive calls— and returns a value, usually in a register. The compilation of a procedure could be:

1. Allocate storage on the stack for compiler-generated temporaries, and adjust the frame pointer.

2. Generate code for the procedure body.

3. Pop off the temporary locations and arguments. Restore the frame pointer.

4. Pop the return address and transfer control to the caller.

The compilation of a procedure call could be:

1. Generate code to evaluate the arguments.

2. Save in the temporary area all live values in registers.

3. Push the arguments on the stack.

4. Push the return address and jump to the procedure.

This model of compilation is so ingrained that almost all processors provide special instructions to do the call and return. Some provide special instructions to allocate a stack frame and save the frame pointer. Some even provide "register windows" that serve to restrict the usage of high-speed register memory to compilers using this model. We contend that there is little to be gained from this view of compilation and a great deal to be lost, both in clarity and in the efficiency of generated code!

An alternative view of compilation is offered by CPS conversion. A procedure call (or return) is just a *goto* that passes parameters [Steele 76] [Steele 76b] [Steele 77], where one or more of the arguments may be continuations. The distinction between compiler-generated temporaries and user variables is discarded, as is the frame pointer, which was redundant with the stack pointer anyway. Generating code for a procedure becomes:

- Generate code for the procedure body and call the continuation.

and for a procedure call:

- Move the arguments into registers and transfer control.

The code generator in Orbit, through adopting a style different from the above, is different from others because all of the uses choices that fit together into a coherent whole:

- Parameters to calls are passed in registers.
- Register allocation and code generation are done at the same time.
- Closure variables are referenced by treating environment pointers just like other variables (using a lazy display).
- The *call* is a fundamental concept and any call may have any number of arguments.

The details of these design decisions are discussed in the remainder of this chapter.

5.2 Splits and Joins

The code generation phase is a simple walk over the node tree, generating code for each procedure call, the goal being to simulate the dynamic behavior of the program. The walk would be trivial (register allocation aside) were it not for the presence of *splits* and *joins* in the code. A split is represented by a call-node with more than one continuation. When the code generator encounters such a node, it must choose which continuation to generate code for first. Using conventional compilation techniques, the choice of which branch of a conditional to generate code for first is not important because all decisions which could be affected by the choice (register assignments) have already been made. When register allocation and code generation are being done at the same time, however, order does make a difference because the branches may eventually join. It may be useful to have *the same variable in different locations on different branches.*

A *join point* is a lambda-node in the node tree that can be arrived at by more than one path. Such a join point is represented by a lambda-node that is bound to a variable. Paths through the code join together at such a point by a call to the variable. At each call to such a variable, the actual parameters and free variables of the bound lambda can be in arbitrary machine locations, but when control is transferred they must all be synchronized; i.e., the actual parameters and free variables must be moved into the same locations on all paths before control is transferred to the join point. This is done by having the first path to arrive at the join point during code generation determine where the arguments to the join will be passed and, it the join point was assigned strategy *registers*, to which registers the free variables should be assigned. In fact, the first time the code generator reaches a join point, it ignores the fact that other paths through the code may join there. All other paths must then conform to the register assignments made by the first. Of course, the information about assignments is used by the code generator as it generates code for the other paths.

Since, for all paths but the first, it may be necessary to insert move instructions to synchronize the registers at join points, the choice of which path to generate code for first is an important one. That choice is made by using a form of *trace scheduling* [Fisher 81, Ellis 85], in which the code generator tries to simulate the dynamic behavior of the program by choosing the branch most likely to be taken when the program is executed. This information could be provided by the programmer, but usually some simple heuristics are used, the most obvious being that a branch which is a call to a “loop” is more likely to be taken than a branch which returns a value (invocation of an unknown continuation). Such information is often not available and other heuristics must be used. A useful heuristic is to calculate the “harm” that is likely to be caused by a wrong choice. For example, if one branch were a sequence of open-coded operations and the other contained a call to an unknown procedure, the open-coded branch should be examined first since an unknown procedure call would force all values out of registers before the join point was reached. If all values are in memory when the join point is reached, there is no useful information that depends on the fact that the particular branch was taken. On the other hand, much can be gained by generating code for the path with open-coded operations while ignoring the other paths.

5.3 Register Allocation

In ORBIT, register allocation is a part of code generation and is a mechanism for answering several questions that the code generator asks:

- Given a procedure call, how are the arguments forced to be in the registers where the procedure expects them to be?

- Given a procedure to be coded in-line, how are the arguments made addressable by machine instructions and where should the result be placed?

- For procedures that are not required to use the standard calling convention, which registers should its formal parameters (and possibly its free variables) be passed in?

- Which machine register should be “spilled” if one is needed but none are available?

In this section the mechanisms that answer these questions will be described.

5.3.1 Parallel Assignment

In the simplest case, generating code for function calls is simply a matter of shuffling the arguments until each argument is in the register where its corresponding formal parameter expects it to be. We refer to this as *parallel assignment*. The problem is that all the registers might be receiving values, so care must be taken in register usage. The one exception is the register called AN. The closure analysis ensures that all closures appearing in a call, which need to be heap-allocated, are merged into one actual runtime structure. The internal routine that allocates the space for this closure returns the structure in the register AN. In the algorithm to be presented, the environment variables for closures with strategy *registers* are treated as if they were arguments.

The shuffling algorithm is straightforward, and will generate little or no code if the register allocation has been good. If there are n arguments being passed we have n pairs of register assignments to perform. These assignments can be accomplished using at most one extra location. If AN (the rootable register that is not used for argument passing) is not being used to hold a closure, it can be the extra location, otherwise a memory location is used. The steps are:

1. Remove all pairs in which the source and target are the same.

2. Find all pairs in which the target is not the same as the source of any other pair and generate the move instruction.

3. Take one of the remaining pairs and look at the target which must match the source of some other pair. If the target of the new pair is not the source of some other, generate move instructions for both pairs. If it is, continue this process of building up a chain.

4. If a cycle is formed, break it by moving one of the registers involved in the cycle into the extra location and generate moves for the others. Then

generate a move from the extra location where it should go.

The assumption being made is that all arguments to a procedure are in registers. Unfortunately, this is only true in some instances. In general, arguments may be in several different types of locations; more specifically, the argument may be:

- In a register.
- A value that can be referenced as part of a machine instruction (e.g. a fixnum).
- A closure that might be in a register or might be internal to another closure.
- In the environment of a closure, but not in any register.

How the first case is handled has already been explained. The second case is not a problem because arguments that are pulled from the instruction stream can be moved into their registers just before jumping to the procedure. The third case, where the argument is a closure, is handled in one of two ways. If the closure has just been allocated, it is either in the register AN or is at some offset from that register. This case can be treated exactly like the immediate operand case. The hard case is when the closure is internal to the global environment. Just like the last case, a value must be fetched from a closure. The difficulty is that the registers needed to do indirection are the same registers to which arguments are being passed. How is it ensured that any are available?

This is the most serious flaw in the *lazy display* strategy. In the worst case we can at least ensure that there will be one machine register available to use in environment chaining: if a continuation is being invoked, the P (procedure) register is available, otherwise it can be made available by making sure that the procedure is the last value moved into its register. In this worst-case situation a large number of move instructions might be generated to accomplish the parallel assignment.

This kind of poor worst-case behavior may be expected when use of a resource is decided dynamically. In this case we want to use machine registers that are holding pointers both to pass arguments and to hold a display. When the number of machine registers available for this is not much greater than the average number of arguments passed to a procedure, these problems can arise. In practice, since most procedures take only a few arguments, a machine with 32 general purpose registers would be more than adequate to prevent the collapse of the display strategy.

The process of parallel assignment is further complicated by the fact that variables can have non-standard representations. As an example, suppose **x** is known to be a fixnum and is being represented as a machine number (tag has been shifted out). If **x** is the first argument to an unknown procedure, the tag will have to be shifted onto **x** before moving it to the first argument register. So in general, each register in the shuffling algorithm is really a register and a representation. A conversion has to be done if the source and target representations are different.

5.3.2 Value Lookup

Each operation that will be open-coded addresses its operands. Part of generating a machine instruction is generating machine references to the operands, which can be in several different machine locations. If a value is to be part of the instruction stream (e.g. a fixnum) there is nothing to do, otherwise there are three possibilities for referencing the value, in order of increasing cost:

1. Fetch it from of a machine register.

2. Fetch it from a register that is actually memory.

3. Fetch it from the environment of a closure.

In the first two cases no extra instructions need to be generated. To get a value from an environment, links are followed from the current environment to the innermost environment holding the value, i.e. use the lazy display. If a pointer to that innermost environment is in a machine register, we can indirect through that register. If the pointer is not in a register, it must be brought into a register. We look at the next innermost link to see if it is in a register, and so on. Eventually the environment pointer for the current closure will be reached; the compiler guarantees that any environment pointer is reachable if a value from the environment it points to may be needed.

The continuation variables introduced during CPS conversion were ignored in the live variable analysis described in Chapter 4 and are fetched differently from other values. A continuation variable can only appear in call position or as a continuation argument. Referencing a continuation variable normally involves restoring the continuation register (stack pointer) to what it was when the procedure binding the variable was invoked. The exception is when the continuation is being invoked from a closure with *stack/recursion*. In this case an instruction must be emitted to test the value in the continuation register. If the continuation register contains a stack-allocated environment it is removed, exposing the continuation to which the continuation variable is really bound.

5.3.3 Generating In-line Code

The only code generation information wired into the compiler is for compiling procedure calls, generating variable references, and allocating closures. All other information is obtained from the early-binding environment passed as an argument to the compiler. Information about generating in-line code for a procedure is contained in primops. A typical definition for the MC68000 processor might be:

```
(define-constant fixnum-logand
  (primop fixnum-logand ()
    ((primop.generate self node)
     (generate-logical-and node))
    ((primop.arg-specs self) '(data data))))
```

which describes the bitwise logical **and** operation.

In the default early-binding environment, **fixnum-logand** will be bound to a primop object that tells the compiler everything known about generating code for bitwise logical and. When the code generator encounters a call to a primop, it calls **primop.generate** on the primop and the call-node instead of generating code for a procedure call. In this case a call to **generate-logical-and** will occur. As required by the 68000, this routine will force one of the argments into a Data register and the other into anything but an Address register and then generate an **and** instruction. It chooses the target register for the **and** instruction by examining the references to the variable being bound to the result of the call and the locations of the arguments. The **primop.arg-specs** clause of the primop definition tells the code generator to try to compute the arguments to this operation in Data registers so that **generate-logical-and** can satisfy the 68000 constraint without extra move instructions.

5.3.4 Register Assignment for Join Points

When a call to a join point is reached for the first time, the variables representing its formal parameters and (if the join point was assigned strategy *registers*) its free variables must be assigned to registers. The registers are assigned using heuristics based on trace scheduling, frequency of use information gathered for variables, and the locations of the actual parameters and free variables at the point of the first call. When a machine register is needed during code generation and none are free, similar heuristics are used to choose a register to be spilled into memory.

5.4 Assembly

Orbit has its own assembler, written in T, that provides communication between the assembler and code generator. The code generator, modulo machine restrictions, emits blocks of code in an order determined by the trace information it has collected. But on the VAX, for example, branch displacements are limited to an eight bit, signed field. The assembler may have to introduce extra jump instructions and rearrange certain code blocks [Szymanski 78]. It uses information provided by the code generator in order to make the best choices for ordering code blocks. The most useful information provided by the code generator is that a particular branch is or is not to a "loop top". This information allows the assembler to order blocks so that all loops are entered by a jump to the end-of-loop test, reducing the size of all loops by one instruction.

A call to Orbit returns two objects: a compiled-code object and an early-binding object. As described in Chapter 3, the early-binding object contains information about variables, defined during the compilation, that may be used by other compilations. A compiled-code object contains a code vector and a data section descriptor. The code section is a bit-vector, returned by the assembler, that contains the instructions stream. The data section descriptor tells the linker (static or dynamic) what values the environment for the code should contain. This compiled-code object can be linked into the scheme system at once, or may be dumped into an "object file" to be linked into a scheme system at some other time.

5.5 Order of Evaluation Revisited

With an understanding of how register allocation and code generation are performed, we can look more closely at the heuristics used to order subforms of calls during CPS conversion.

Remember that in the standard calling convention, arguments to procedures are passed in a set of registers and values are returned in those same registers. The following discussion refers to source code before CPS conversion. Orbit uses a few simple heuristics to choose an order for evaluation of subforms:

1. It is better to first evaluate the subforms that will cause all values to be forced out of registers, unknown procedure calls being the obvious examples. If an in-line procedure call were done first, the result would just have to be saved on the stack prior to the unknown call.

2. When choosing between subforms that are unknown calls, it is better to choose an order that is likely to cause a returned value to be in the same register where it will be used as an argument to another call. Move instructions may be avoided by making a good choice.

3. When choosing between subforms that will cause in-line code to be generated, information about data-dependency can be used.

In the following examples we assume that all arithmetic is on fixnums. The second case is well illustrated by the infamous benchmark procedure TAK:

```
(define (tak x y z)
    (if (not (< y x))
        z
        (tak (tak (- x 1) y z)       ;;; call #1
             (tak (- y 1) z x)       ;;; call #2
             (tak (- z 1) x y))))    ;;; call #3
```

An order must be chosen for compiling the three recursive calls to **tak**. If we choose either #2 or #3 to do first, the arguments **x**, **y** and **z** will have to be shuffled in their registers, but if we do #1 first no moves are necessary. However, after the first call it makes no difference which call is done next because the arguments will necessarily be on the stack and not in registers. On the other hand, if all other things were equal, it would be better to do the first call last because then the value would be returned in the register where it is being used for the final call to **tak**.

The heuristic is very simple: evaluate first that call in which variable arguments occur in the same position in the call as in the formal parameter list of the lambda that binds them. If there is no preference by this criterion, do the first call last. In this example, the first call has the arguments **y** and **z** in the same positions as they appear in the parameter-list of **tak**.

The third heuristic involves the issue of data dependencies. As an example consider iterative factorial:

```
(define (factorial n)
  (letrec ((fac (lambda (result i)
                  (if (= i 0)
                      result
                      (fac (* result i) (- i 1))))))
    (fac 1 n)))
```

It would be most efficient to have **result** and **i** sit in registers throughout the iteration. For that to happen the multiplication subform would have to be generated before the subtraction because the subtraction will destroy the value of **i** needed for the multiplication. In this heuristic we look at the lambda to which **fac** is bound. The formal parameters of this lambda are **result** and **i**, so we have the parallel assignment equations: $result \leftarrow result * i$ and $i \leftarrow i - 1$. If there is a parallel assignment equation whose left hand side is not used on the right hand side of some other equation in the same set, that call whould be ordered first. In this example **result** is not used by the second equation but **i** is used by the first equation, thus the multiplication is done first.

Chapter 6

Examples and Benchmarks

To get a feel for what is produced by the closure analysis and code generation algorithms discussed in the previous chapters, several examples will be presented in this chapter. Following that the results of two sets of benchmarks will be presented, comparing Orbit running in T3 with commercial Pascal and Common Lisp systems.

6.1 Examples

Though Orbit generates code for the MC68000, Vax, and NS32032 processors, a pseudo assembly language will be used for the examples. There will be two examples of strategy *registers*, one recursive and the other iterative, and an example of *stack/loop*. Several symbols in the pseudo code need explanation (assume that N is the number of arguments passed to a procedure or continuation):

NARGS A register that holds $N + 1$ for a procedure call and $-(N + 1)$ for a call to a continuation. This number is used to indicate the number of arguments supplied to an n-ary procedure. It is also used to do number of argument checking if desired.

NIL A register that always contains the empty list, which is the same as the false value.

PEA The pseudo-instruction for push effective address.

P,A1,A2,.. The argument registers in the standard calling sequence.

CONTINUE Invoke the continuation pointed to by the stack pointer.

CALL Call the procedure in the P register.

The examples ultimately expand into **letrec** and only that part of each procedure will be shown. First the procedure will be presented, then the CPS code, and finally the CPS code with pseudo assembly code intermixed. In the CPS code, the first two arguments to **if** are the succeed and fail continuations.

6.1.1 Strategy *registers* (iterative)

As an example of strategy *registers* in the iterative case we use the MEMQ function:

```
(define (memq obj list)
  (letrec ((memq
    (lambda (list)
      (cond ((null? list) '())
            ((eq? obj (car list)) list)
            (else (memq (cdr list)))))))
    (memq list)))
```

After CPS conversion:

```
MEMQ: (lambda (k list) (if L1 L2 null? list))
L1:     (lambda () (k '()))
L2:      (lambda () (car L3 list))
L3:        (lambda (v) (if L4 L5 eq? obj v))
L4:          (lambda () (k list))
L5:          (lambda () (cdr L6 list))
L6:            (lambda (w) (memq k w))
```

The lambda bound to **memq** is assigned strategy *registers*. The environment consists of **obj**. On entry to **memq**, **list** is allocated to A1 and **obj** is allocated to A2. The generated code is as follows:

```
MEMQ: (lambda (k list) (if L1 L2 null? list))

  CMP A1,NIL
  BNE L2

L1:      (lambda () (k '()))

  MOVE NIL,A1     * move nil to first argument register
  MOVE #-2,NARGS  * one argument to continuation
  CONTINUE

L2:      (lambda () (car L3 list))

                  * (car LIST) delayed

L3:        (lambda (v) (if L4 L5 eq? obj v))

  CMP CAR(A1),A2  * (eq? (car LIST) OBJ)
  BNE L5

L4:          (lambda () (k list))

                  * LIST is already in A1
  MOVE #-2, NARGS * one argument to continuation
  CONTINUE
```

```
L5:          (lambda () (cdr L6 list))

  MOVE CDR(A1),A1 * LIST (in A1) is dead after the cdr

L6:            (lambda (w) (memq k w))

  JUMP MEMQ       * this jump is eliminated by the assembler
```

6.1.2 Strategy *registers* (recursive)

As an example of strategy *registers* in the genuinely recursive case we use the DELQ! function:

```
(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)))
```

After CPS conversion:

```
DELQ!: (lambda (k list) (if L1 L2 null? list))
L1:      (lambda () (k '()))
L2:      (lambda () (car L3 list))
L3:        (lambda (v) (if L4 L5 eq? obj v))
L4:        (lambda () (cdr L6 list))
L6:          (lambda (v) (delq! k v))
L5:        (lambda () (cdr L7 list))
L7:          (lambda (v) (delq! L8 v))
L8:            (lambda (w) (set-cdr L9 list w))
L9:              (lambda ignore (k list))
```

The lambda bound to **delq!** is assigned strategy *registers*. The environment consists of **obj**. On entry to **delq!**, **list** is allocated to A1 and **obj** is allocated to A2. The generated code is as follows:

```
DELQ!: (lambda (k list) (if L1 L2 null? list))

  CMP A1,NIL
  BNE L2

L1:      (lambda () (k '()))

  MOVE NIL,A1     * move nil to first argument register
  MOVE #-2,NARGS  * one argument to continuation
  CONTINUE

L2:      (lambda () (car L3 list))

                  * (car LIST) delayed

L3:        (lambda (v) (if L4 L5 eq? obj v))

  CMP CAR(A1),A2  * (eq? (car list) obj)
  BNE L5

L4:        (lambda () (cdr L6 list))

  MOVE CDR(A1),A1 * LIST (in A1) is dead after cdr

L6:          (lambda (v) (delq! k v))

  JUMP DELQ!      * this jump is eliminated by the assembler
```

```
L5:        (lambda () (cdr L7 list))

  PUSH A1         * save LIST on the stack
  PEA L8          * continuation L8 has been hoisted to here
  MOVE CDR(A1),A1 * LIST (in A1) is dead after cdr

L7:          (lambda (v) (delq! L8 v))

  JUMP DELQ!

L8:            (lambda (w) (set-cdr L9 list w))

  MOVE LIST(SP),P * get LIST into a register
  MOVE A1,CDR(P)  * set (cdr LIST) to returned value

L9:              (lambda ignore (k list))

  MOVE P,A1       * move LIST to first argument register
  ADD #8,SP       * remove continuation L8 from stack
  MOVE #-2,NARGS  * one argument to continuation
  CONTINUE
```

6.1.3 Strategy *stack/loop*

As an example of *stack/loop* we use the MEM function:

```
(define (mem pred obj list)
  (letrec ((mem
    (lambda (list)
      (cond ((null? list) '())
            ((pred obj (car list)) list)
            (else (mem (cdr list)))))))
    (mem list)))
```

```
MEM: (lambda (k list) (if L1 L2 null? list))
L1:    (lambda () (k '()))
L2:    (lambda () (car L3 list))
L3:      (lambda (v) (pred L4 obj v))
L4:        (lambda (b) (if L5 L6 true? b))
L5:          (lambda () (k list))
L6:          (lambda () (cdr L7 list))
L7:            (lambda (w) (mem k w))
```

The lambda bound to **mem** is assigned *stack/loop*. The environment consists of **obj** and **pred**. Before entering **mem**, **pred** and **obj** are pushed on the stack, along with a header describing the closure. On entry to **mem**, **list** is allocated to A1. The generated code is as follows:

```
MEM: (lambda (k list) (if L1 L2 null? list))

  CMP A1,NIL
  BNE L2

L1:      (lambda () (k '()))

  MOVE NIL,A1     * move nil to first argument register
  ADD #12,SP      * remove environment (pred,obj,header)
  MOVE #-2,NARGS  * one argument to continuation
  CONTINUE

L2:    (lambda () (car L3 list))

  PUSH A1         * save LIST on stack
  PEA L4          * L4 has been hoisted to here
  MOVE CAR(A1),A2 * (car LIST) into A2 for second arg

L3:      (lambda (v) (pred L4 obj v))

  MOVE OBJ(SP),A1 * OBJ is first argument
  MOVE PRED(SP),P * PRED is procedure
  MOVE #3,NARGS   * two arguments to procedure
  CALL
```

```
L4:        (lambda (b) (if L5 L6 true? b))

  CMP A1,NIL      * compare result to nil
  BEQ L6          * if equal the true? test fails

L5:          (lambda () (k list))

  MOVE LIST(SP),A1 * move LIST into first argument register
  ADD #20,SP       * remove loop environment (12) and L4 (8)
  MOVE #-2,NARGS   * one argument to continuation
  CONTINUE

L6:          (lambda () (cdr L7 list))

  MOVE LIST(SP),A1 * get LIST into a register
  MOVE CDR(A1),A1

L7:            (lambda (w) (mem k w))

  ADD #8,SP        * remove continuation L4
  JUMP MEM         * this jump is removed by the assembler
```

6.2 Benchmarks

In some sense, the assembly code produced by a compiler is the best way to consider its performance. The code in the examples shown is pretty close to optimal. It is also useful to compare the performance with other systems. In particular, since the thesis was that a Scheme compiler can produce code competitive with

Pascal, Orbit should be benchmarked against a good Pascal compiler. It was also benchmarked against a Common Lisp compiler.

6.2.1 Pascal

The Pascal benchmarks were taken from a set of Modula-2 benchmarks used at DEC Western Research Laboratory [Powell 84]. The benchmarks were translated into Pascal from Modula-2 and are in Appendix A. They were then translated into T, and appear in Appendix B. These benchmarks were run on an Apollo DN3000 with 4 megabytes of memory. The Apollo Pascal Compiler release 9.2.3 was used, which is Apollo's standard optimizing compiler (Apollo relies heavily on Pascal since most of its operating system is written in it).

Benchmarking fairly is difficult, but every effort was made to be fair including the following: the T code was compiled without number of argument checking and it was allowed to assume that procedures would not be redefined. In addition, Orbit was not allowed to integrate procedures unless they were declared as local procedures in the Pascal code. The results of the benchmarks were (time is in seconds):

Benchmark	Pascal	T
perm	.95	.69
towers	1.24	1.03
quick	.35	.26
fib	.17	.12
tak	.57	.30
bubble	.73	.67
puzzle	3.17	3.33
intmm	2.41	2.45

The comparisons fall into two groups. The bubble, puzzle, and intmm benchmarks have no recursive procedure calls. The results are similar for both T and Pascal, indicating that for iterative code the performance is comparable. When there are lots of procedure calls in the Pascal code, T does considerably better. This is because it is much better at compiling procedure calls, where an iteration is just a special kind of procedure call.

6.2.2 Common Lisp

It also seems reasonable to benchmark against a Common Lisp system. I used the published [Sun 86] timings on the Gabriel benchamrks [Gabriel 85] for Lucid Common Lisp running on a Sun 3/160M with 12 megabytes of memory and code generated for the MC68020. The T code was run on a Sun 3/160 with 4 megabytes of memory and code generated for the MC68000 only. The results were:

Benchmark	Lucid CL	T
tak	.44	.22
puzzle	7.70	2.40
triangle	131.72	79.18
idiv2	.92	.50
rdiv2	1.46	.76
destru	2.14	.98
deriv	3.68	2.44
dderiv	5.44	3.02
fprint	1.56	1.80
fread	4.20	3.08
tprint	1.94	1.46
curry	1.1	.40
kons	2.20	.94

The Common Lisp times are a little faster than they should be because of the use of MC68020 instructions as reported with the published figures. Several of the Gabriel benchmarks were left out because they use features which have not yet been implemented efficiently in T, but are implemented efficiently in production systems. In addition, Orbit does not generate in-line floating point instructions at the present time. One way in which the Gabriel benchmarks are deficient is that none of them have any closures which must be heap-allocated. The last two benchmarks, curry and kons, were used to check performance on closures. The code for these two is in Appendix C.

Chapter 7

Final thoughts

It has been demonstrated that it is possible to implement Scheme as efficiently as Pascal. One obvious benefit of this fact is that many programmers can choose to use Scheme to do real programming without worrying about the performance being poor.

This work also has several immediate applications:

Numerical Computation. Lisp and Scheme have normally been associated with symbolic computation but the same techniques which have been used in Orbit can be easily extended to do floating-point computations efficiently. In the near furture it is entirely possible that Fortran programmers could use Scheme for scientific computation.

Functional Languages. Work in functional languages and would-be users of functional languages have been impeded by the fact that efficient implementations do not exist. Orbit can serve as the back end of a compiler for a lazy functional language doing strictness analysis and other optimizations for efficient execution [Hudak 84, Hudak 84b].

Parallel Scheme. Orbit can be the back end for a compiler generating code for a MIMD machine with a front end taking care of the static part of task distribution. In such a system good sequential performance on the individual processors is obviously important.

Appendix A

Benchmarks in Pascal

```
program bench(INPUT, OUTPUT);

(*  This is a suite of benchmarks that are relatively short,
    both in program size and execution time.  It requires no
    input, and prints the execution time for each program, using
    the system- dependent routine Getclock, below, to find out
    the current CPU time.  It does a rudimentary check to make
    sure each program gets the right output.  These programs were
    gathered by John Hennessy and modified by Peter Nye.
 *)

const
    (* Towers *)
    maxcells = 18;

    (* Puzzle *)
    size = 511;
    classmax = 3;
    typemax = 12;
    d = 8;

    (* Bubble, Quick *)
    sortelements = 5000;
    srtelements = 500;

type
    (* Perm *)
    permrange = 0 .. 10;

    (* Towers *)
    discsizrange = 1..maxcells;
    stackrange = 1..3;
    cellcursor = 0..maxcells;
    element =
        record
            discsize:discsizrange;
            nxt:cellcursor;
        end;
    emsgtype = string (* %PACKED *) {array[1..15] of char};
    (* Puzzle *)
    piececlass = 0..classmax;
    piecetype = 0..typemax;
    position = 0..size;

    (* Bubble, Quick *)
    listsize = 0..sortelements;
    lstsize = 0..srtelements;
    sortarray = array [listsize] of integer32;
    srtarray = array [lstsize] of integer32;

var
    (* global *)
    timer: integer32;
    xtimes: array[1..10] of integer32;
    i,j,k:integer32;
    seed: integer32;

    (* Perm *)
    permarray: array [permrange] of permrange;
    pctr: 0..65535;

    (* Towers *)
    stack: array [stackrange] of cellcursor;
    cellspace: array[1..maxcells] of element;
    freelist: cellcursor;
    movesdone: integer32;


    (* Puzzle *)
    piececount: array [piececlass] of 0..13;
    class: array [piecetype] of piececlass;
    piecemax: array [piecetype] of position;
    puzzl: array [position] of boolean;
    p: array [piecetype,position] of boolean;
    m,n: position;
    kount: integer32;

    (* Bubble, Quick *)
    sortlist: sortarray;
    srtlist: srtarray;
    biggest, littlest: integer32;
    bggest, lttlest: integer32;
    top: lstsize;

procedure writef( foo: text; s: string; num: integer32);
begin
    write( s );
    writeln;
    write( num/100:5:2);
    writeln;
end;

(* global procedures *)
```

```
procedure initclock; EXTERN;
procedure stopclock; EXTERN;

function getclock : integer32; EXTERN;

procedure Initrand;
    begin
    seed := 74755;
    end {Initrand};

function Rand : integer32;
    begin
    seed := (seed * 1309 + 13849) mod 65536;
    rand := seed;
    end {Rand};


procedure Perm;

    (* Permutation program, heavily recursive,
       written by Denny Brown. *)

    procedure Swap(var a,b : permrange);
        var t : permrange;
        begin
        t := a;  a := b;  b := t;
        end {Swap};

    procedure Initialize;
        var i : permrange;
        begin
        for i := 1 to 7 do begin
            permarray[i]:=i-1;
             end;
        end {Initialize};

    procedure Permute(n : permrange);
        var k : permrange;
        begin   (* permute *)
        pctr := pctr + 1;
        if n<>1 then BEGIN
            Permute(n-1);
            for k := n-1 downto 1  do begin
                Swap(permarray[n],permarray[k]);
                Permute(n-1);
                Swap(permarray[n],permarray[k]);
                 end;
             end;
        end {Permute};     (* permute *)

    begin   (* Perm *)
    pctr := 0;
    for i := 1 to 5 do begin
        Initialize;
        Permute(7);
```

```
         end;
    if pctr <> 43300 then BEGIN
        writef (output, ' Error in Perm.\n',0); end;
    end {Perm};     (* Perm *)



procedure Towers;

    (*  Program to Solve the Towers of Hanoi *)

    procedure Error (emsg:emsgtype);
        begin
{        writef (output, ' Error in Towers: %s\n', emsg); }
        writef( output, ' Error in Towers: ',0 );
        writef( output, emsg,0 );
        writeln;
        end {Error};

    procedure Makenull (s:stackrange);
        begin
        stack[s]:=0;
        end {Makenull};

    function Getelement :cellcursor;
        var nxtfree:cellcursor;
        begin
        if freelist>0 then BEGIN

            nxtfree := freelist;
            freelist:=cellspace[freelist].nxt;
            getelement := nxtfree;

        END else BEGIN
            Error('out of space   '); end;
        end {Getelement};

    procedure Push(i:discsizrange;s:stackrange);

        var
            errorfound:boolean;
            localel:cellcursor;
        begin

        errorfound:=false;
        if stack[s] > 0 then BEGIN
            if cellspace[stack[s]].discsize<=i then BEGIN

                errorfound:=true;
                Error('disc size error');
                 end; end;
        if not errorfound then BEGIN

            localel:=Getelement;
            cellspace[localel].nxt:=stack[s];
        stack[s]:=localel;
        cellspace[localel].discsize:=i;

     end; end {Push};

procedure Init (s:stackrange;n:discsizrange);

    var
        discctr:discsizrange;
    begin
    Makenull(s);
    for discctr:=n downto 1  do begin
        Push(discctr,s); end;
    end {Init};

function Pop (s:stackrange):discsizrange;

    var
        popresult:discsizrange;
        temp:cellcursor;
    begin
    if stack[s] > 0 then BEGIN

        popresult:=cellspace[stack[s]].discsize;
        temp:=cellspace[stack[s]].nxt;
        cellspace[stack[s]].nxt:=freelist;
        freelist:=stack[s];
        stack[s]:=temp;
        pop := popresult;

    END else BEGIN
        Error('nothing to pop '); end;
    end {Pop};

procedure Move (s1,s2:stackrange);
    begin

    Push(Pop(s1),s2);
    movesdone:=movesdone+1;
    end {Move};

procedure Towers(i,j,k:integer32);
    var other:integer32;
    begin

    if k=1 then BEGIN
        Move(i,j);
    END else BEGIN

        other:=6-i-j;
        Towers(i,other,k-1);
        Move(i,j);
        Towers(other,j,k-1);
         end;
    end {Towers};
    begin (* Towers *)
    for freelist:=1 to maxcells do begin
        cellspace[freelist].nxt:=freelist-1; end;
    freelist:=maxcells;
    Init(1,14);
    Makenull(2);
    Makenull(3);
    movesdone:=0;
    Towers(1,2,14);
    if movesdone <> 16383 then BEGIN
        writef (output, ' Error in Towers.\n',0); end;
    end {Towers}; (* Towers *)

procedure Puzzle;

    (* A compute-bound program from Forest Baskett. *)

    function Fit (i : piecetype; j : position) : boolean;

        var     k       :       position;
        label   999;
        begin
        for k := 0 to piecemax[i] do begin
            if p[i,k] then BEGIN if puzzl[j+k] then
                BEGIN fit := false; goto 999 end; end; end;
        fit := true;
        999:
        end {Fit};

    function Place (i : piecetype; j : position) : position;

        var     k       :       position;
        label   999;

        begin
        for k := 0 to piecemax[i] do begin
            if p[i,k] then BEGIN puzzl[j+k] := true; end; end;
        piececount[class[i]] := piececount[class[i]] - 1;
        for k := j to size do begin
            if not puzzl[k] then BEGIN place := (k);
                                   goto 999 end; end;
        place := (0);
        999:
        end {Place};

    procedure Remove (i : piecetype; j : position);

        var     k       :       position;

        begin
        for k := 0 to piecemax[i] do begin
            if p[i,k] then BEGIN puzzl[j+k] := false; end; end;
        piececount[class[i]] := piececount[class[i]] + 1;
    end {Remove};

function Trial (j : position) : boolean;


    var     i       :       piecetype;
        k       :       position;
    label 999;

    begin
    kount := kount + 1;
    for i := 0 to typemax do begin
        if piececount[class[i]] <> 0 then BEGIN
            if Fit (i, j) then BEGIN
                k := Place (i, j);
                if Trial(k) or (k = 0) then BEGIN
                    trial := (true);
                    goto 999;
                END else BEGIN Remove (i, j); end;
                 end; end; end;
    trial := (false);
    999:
    end {Trial};

begin
for m := 0 to size do begin puzzl[m] := true; end;
for i := 1 to 5 do begin for j := 1 to 5 do
  begin for k := 1 to 5 do begin
    puzzl[i+d*(j+d*k)] := false; end; end; end;
for i := 0 to typemax do begin for m := 0 to size do
  begin p[i, m] := false end  end;
for i := 0 to 3 do begin for j := 0 to 1 do
  begin for k := 0 to 0 do begin
    p[0,i+d*(j+d*k)] := true; end; end; end;
class[0] := 0;
piecemax[0] := 3+d*1+d*d*0;
for i := 0 to 1 do begin for j := 0 to 0 do
  begin for k := 0 to 3 do begin
    p[1,i+d*(j+d*k)] := true; end; end; end;
class[1] := 0;
piecemax[1] := 1+d*0+d*d*3;
for i := 0 to 0 do begin for j := 0 to 3 do
  begin for k := 0 to 1 do begin
    p[2,i+d*(j+d*k)] := true; end; end; end;
class[2] := 0;
piecemax[2] := 0+d*3+d*d*1;
for i := 0 to 1 do begin for j := 0 to 3 do
  begin for k := 0 to 0 do begin
    p[3,i+d*(j+d*k)] := true; end; end; end;
class[3] := 0;
piecemax[3] := 1+d*3+d*d*0;
for i := 0 to 3 do begin for j := 0 to 0 do
  begin for k := 0 to 1 do begin
    p[4,i+d*(j+d*k)] := true; end; end; end;
class[4] := 0;
```

```
piecemax[4] := 3+d*0+d*d*1;
for i := 0 to 0 do begin for j := 0 to 1
  do begin for k := 0 to 3 do begin
    p[5,i+d*(j+d*k)] := true; end; end; end;
class[5] := 0;
piecemax[5] := 0+d*1+d*d*3;
for i := 0 to 2 do begin for j := 0 to 0 do
  begin for k := 0 to 0 do begin
    p[6,i+d*(j+d*k)] := true; end; end; end;
class[6] := 1;
piecemax[6] := 2+d*0+d*d*0;
for i := 0 to 0 do begin for j := 0 to 2 do
  begin for k := 0 to 0 do begin
    p[7,i+d*(j+d*k)] := true; end; end; end;
class[7] := 1;
piecemax[7] := 0+d*2+d*d*0;
for i := 0 to 0 do begin for j := 0 to 0 do
  begin for k := 0 to 2 do begin
    p[8,i+d*(j+d*k)] := true; end; end; end;
class[8] := 1;
piecemax[8] := 0+d*0+d*d*2;
for i := 0 to 1 do begin for j := 0 to 1 do
  begin for k := 0 to 0 do begin
    p[9,i+d*(j+d*k)] := true; end; end; end;
class[9] := 2;
piecemax[9] := 1+d*1+d*d*0;
for i := 0 to 1 do begin for j := 0 to 0 do
  begin for k := 0 to 1 do begin
    p[10,i+d*(j+d*k)] := true; end; end; end;
class[10] := 2;
piecemax[10] := 1+d*0+d*d*1;
for i := 0 to 0 do begin for j := 0 to 1 do
  begin for k := 0 to 1 do begin
    p[11,i+d*(j+d*k)] := true; end; end; end;
class[11] := 2;
piecemax[11] := 0+d*1+d*d*1;
for i := 0 to 1 do begin for j := 0 to 1 do
  begin for k := 0 to 1 do begin
    p[12,i+d*(j+d*k)] := true; end; end; end;
class[12] := 3;
piecemax[12] := 1+d*1+d*d*1;
piececount[0] := 13;
piececount[1] := 3;
piececount[2] := 1;
piececount[3] := 1;
m := 1+d*(1+d*1);
kount := 0;
if Fit(0, m) then BEGIN n := Place(0, m);
END else BEGIN writef (output, 'Error1 in Puzzle\n',0); end;
if not Trial(n) then
   BEGIN writef (output, 'Error2 in Puzzle.\n',0); END
else BEGIN if kount <> 2005 then BEGIN writef
    (output, 'Error3 in Puzzle.\n',0); end; end;
end {Puzzle};
```

```
procedure qInitarr;
        var i:integer32;
        begin
        Initrand;
        biggest := 0; littlest := 0;
        for i := 1 to sortelements do begin

            sortlist[i] := Rand mod 100000 - 50000;
            if sortlist[i] > biggest then
               BEGIN biggest := sortlist[i];
            END else BEGIN if sortlist[i] < littlest then
                   BEGIN littlest := sortlist[i]; end; end;
             end;
        end {Initarr};


procedure Quick;

    (* Sorts an array using quicksort *)


    procedure Quicksort(var a: sortarray; l,r: listsize);
        (* quicksort the array A from start to finish *)
        var i,j: listsize;
            x,w: integer32;
        begin
        i:=l; j:=r;
        x:=a[(l+r) DIV 2];
        repeat
            while a[i]<x do begin i := i+1; end;
            while x<a[j] do begin j := j-1; end;
            if i<=j then BEGIN
                w := a[i];
                a[i] := a[j];
                a[j] := w;
                i := i+1;    j:= j-1;
                end;
         until i>j;
        if l <j then BEGIN Quicksort(a,l,j); end;
        if i<r then BEGIN Quicksort(a,i,r); end;
        end {Quicksort};


    begin
    Quicksort(sortlist,1,sortelements);
    if (sortlist[1] <> littlest) or (sortlist[sortelements] <> biggest)
       then BEGIN
        writef (output, ' Error in Quick.\n',0); end;
    end {Quick};


procedure bInitarr;
        var i:listsize;
        begin
        Initrand;
        bggest := 0; lttlest := 0;
        for i := 1 to srtelements do begin

            srtlist[i] := Rand mod 100000 - 50000;
            if srtlist[i] > bggest then BEGIN bggest := srtlist[i];
            END else BEGIN if srtlist[i] < lttlest then
                BEGIN lttlest := srtlist[i]; end; end;
             end;
        end {Initarr};

procedure Bubble;
    (* Sorts an array using bubblesort *)


    begin
    top:=srtelements;

    while top>1 do begin

        i:=1;
        while i<integer(top) do begin

            if srtlist[i] > srtlist[i+1] then BEGIN
                j := srtlist[i];
                srtlist[i] := srtlist[i+1];
                srtlist[i+1] := j;
                 end;
            i:=i+1
             end;

        top:=top-1;
         end;
    if (srtlist[1] <> lttlest) or (srtlist[srtelements] <> bggest)
       then BEGIN
        writef (output, 'Error3 in Bubble.\n',0); end;
    end {Bubble};

procedure Fib;

    function lfib (n: integer32) : integer32;
       begin
        if (n = 0) or (n = 1)
           then lfib := 1
           else lfib := lfib(n - 1) + lfib(n - 2);
       end;

begin
 if lfib(20) <> 10946
    then writef (output, 'Error3 in Fib.\n',0);
end{Fib};

procedure Tak;

    function ltak (x: integer32; y: integer32; z: integer32) :
             integer32;
```

```
     begin
      if not (y < x)
         then ltak := z
         else ltak := ltak(ltak(x-1,y,z),ltak(y-1,z,x),ltak(z-1,x,y));
     end;

begin
 if ltak(18,12,6) <> 7
    then writef (output, 'Error3 in Tak.\n',0);
end;


begin
initclock;

Perm;     timer := Getclock; Perm;
          writef (output, 'Perm',timer-Getclock);
Towers;   timer := Getclock; Towers;
          writef (output, 'Towers',timer-Getclock);
Puzzle;   timer := Getclock; Puzzle;
          writef (output, 'Puzzle',timer-Getclock);
qinitarr; Quick;
qinitarr; timer := Getclock; Quick;
          writef (output, 'Quick',timer-Getclock);
binitarr; bubble;
binitarr; timer := Getclock; Bubble;
          writef (output, 'Bubble',timer-Getclock);
Fib;      timer := Getclock; Fib;
          writef (output, 'Fib',timer-Getclock);
Tak;      timer := Getclock; Tak;
          writef (output, 'Tak',timer-Getclock);
writeln;

stopclock;

end {bench}.
```

Appendix B

Benchmarks in T

```
;;; Quick and Bubble

(define-constant sortelements 5000)
(define-constant srtelements 500)

(lset seed 0)
(lset biggest 0)
(lset littlest 0)
(lset bggest 0)
(lset lttlest 0)
(lset top 0)
(declare local seed biggest littlest top bggest lttlest)

(define sortlist (make-vector sortelements))
(define srtlist (make-vector srtelements))

(define (initrand) (set seed 74755))
(define (rand)
  (set seed (mod (fx+ (fx* seed 1309) 13849) 65536))
  seed)

(define (qinitarr)
  (initrand)
  (set biggest 0)
  (set littlest 0)
  (do ((i 0 (fx+ i 1)))
      ((fx>= i sortelements) (return))
    (set (vref sortlist i) (fx- (mod (rand) 100000) 50000))
    (cond ((fx> (vref sortlist i) biggest)
           (set biggest (vref sortlist i)))
          ((fx< (vref sortlist i) littlest)
           (set littlest (vref sortlist i))))))

(define (binitarr)
  (initrand)
  (set bggest 0)
  (set lttlest 0)
  (do ((i 0 (fx+ i 1)))
      ((fx>= i srtelements) (return))
    (set (vref srtlist i) (fx- (mod (rand) 100000) 50000))
    (cond ((fx> (vref srtlist i) bggest)
           (set bggest (vref srtlist i)))
          ((fx< (vref srtlist i) lttlest)
           (set lttlest (vref srtlist i))))))

(define (-quicksort a l r)
  (let ((x (vref a (fx/ (fx+ l r) 2))))
    (iterate loop ((i l) (j r))
      (cond ((fx> i j)
             (if (fx< l j) (-quicksort a l j))
             (if (fx< i r) (-quicksort a i r)))
            (else
             (do ((i i (fx+ i 1)))
                 ((fx>= (vref a i) x)
                  (do ((j j (fx- j 1)))
                      ((fx>= x (vref a j))
                       (cond ((fx<= i j)
                              (exchange (vref a i) (vref a j))
                              (loop (fx+ i 1) (fx- j 1)))
                             (else
                              (loop i j))))))))))))

(define (quick)
  (-quicksort sortlist 0 (fx- sortelements 1))
  (if (or (fxn= (vref sortlist 0) littlest)
          (fxn= (vref sortlist (fx- sortelements 1)) biggest))
      (error "in Quick")))

(define (bubble)
  (set top (fx- srtelements 1))
  (do ((top (fx- srtelements 1) (fx- top 1)))
      ((fx<= top 0)
       (if (or (fxn= (vref srtlist 0) lttlest)
               (fxn= (vref srtlist (fx- srtelements 1)) bggest))
           (error " in Bubble")))
    (do ((i 0 (fx+ i 1)))
        ((fx>= i top) (return))
      (if (fx> (vref srtlist i) (vref srtlist (fx+ i 1)))
          (exchange (vref srtlist i) (vref srtlist (fx+ i 1)))))))

;;; Tak

(define (%tak x y z)
  (labels (((tak x y z)
    (if (not (fx< y x))
        z
        (tak (tak (fx- x 1) y z)
             (tak (fx- y 1) z x)
             (tak (fx- z 1) x y)))))
     (tak x y z)))

(define (tak) (%tak 18 12 6))

;;; Fib
```

```
(define (%fib n)
 (labels (((fib n)
  (cond ((fx= n 0) 1)
        ((fx= n 1) 1)
        (else (fx+ (fib (fx- n 1)) (fib (fx- n 2)))))))
  (fib n)))

(define (fib) (%fib 20))

;;; Puzzle

(define-constant size 511)
(define-constant classmax 3)
(define-constant typemax 12)
(lset index 0)
(lset iii 0)
(lset kount 0)
(define-constant d 8)
(declare local kount iii index)
(define piececount (vector-fill (make-vector (fx+ 1 classmax)) 0))
(define class      (vector-fill (make-vector (fx+ 1 typemax)) 0))
(define piecemax   (vector-fill (make-vector (fx+ 1 typemax)) 0))
(define puzzle-vec (make-vector (fx+ 1 size)))
(define p          (make-vector (fx* (fx+ 1 typemax) (fx+ 1 size))))

(define-integrable (pidx i j) (fx+ (fx* 512 i) j))

(define (puzzle)
  (labels (

((fit i j)
  (let ((end (vref piecemax i)))
    (iterate fit1 ((k 0))
      (cond ((fx> k end) t)
            ((and (vref p (pidx i k))
                  (vref puzzle-vec (fx+ j k)))
             nil)
            (else (fit1 (fx+ k 1)))))))

((place i j)
  (let ((end (vref piecemax i)))
    (do ((k 0 (fx+ k 1)))
        ((fx> k end))
      (if (vref p (pidx i k)) (set (vref puzzle-vec (fx+ j k)) t)))
    (set (vref piececount (vref class i))
         (fx- (vref piececount (vref class i)) 1))
    (iterate place1 ((k j))
      (cond ((fx> k size) 0)
            ((not (vref puzzle-vec k)) k)
            (else (place1 (fx+ k 1)))))))

((puzzle-remove i j)
  (let ((end (vref piecemax i)))
    (do ((k 0 (fx+ 1 k)))
        ((fx> k end))
      (if (vref p (pidx i k)) (set (vref puzzle-vec (fx+ j k))  nil)))
    (set (vref piececount (vref class i))
         (fx+ (vref piececount (vref class i)) 1))))

((trial j)
    (set kount (fx+ kount 1))
    (iterate trial1 ((i 0))
      (cond ((fx> i typemax) nil)
            ((and (not (fx= (vref piececount (vref class i)) 0))
                  (fit i j))
             (let ((k (place i j)))
               (cond ((or (trial k) (fx= k 0)) t)
                     (else (puzzle-remove i j)
                           (trial1 (fx+ i 1))))))
            (else (trial1 (fx+ i 1))))))


((definepiece iclass ii jj kk)
  (set index 0)
    (do ((i 0 (fx+ 1 i)))
        ((fx> i ii))
      (do ((j 0 (fx+ 1 j)))
          ((fx> j jj))
        (do ((k 0 (fx+ 1 k)))
            ((fx> k kk))
          (set index  (fx+ i (fx* d (fx+ j (fx* d k)))))
          (set (vref p (pidx iii index))  t))))
    (set (vref class iii) iclass)
    (set (vref piecemax iii) index)
    (if (not (fx= iii typemax)) (set iii (fx+ iii 1))))
 )

  (do ((m 0 (fx+ 1 m)))
      ((fx> m size))
    (set (vref puzzle-vec m) t))
  (do ((i 1 (fx+ 1 i)))
      ((fx> i 5))
    (do ((j 1 (fx+ 1 j)))
        ((fx> j 5))
      (do ((k 1 (fx+ 1 k)))
          ((fx> k 5))
        (set (vref puzzle-vec (fx+ i (fx* d (fx+ j (fx* d k)))))
             nil))))
  (do ((i 0 (fx+ 1 i)))
      ((fx> i typemax))
      (do ((m 0 (fx+ 1 m)))
          ((fx> m size))
          (set (vref p (pidx i m))  nil)))
  (set iii 0)
  (definePiece 0 3 1 0)
  (definePiece 0 1 0 3)
  (definePiece 0 0 3 1)
  (definePiece 0 1 3 0)
  (definePiece 0 3 0 1)
  (definePiece 0 0 1 3)
  (definePiece 1 2 0 0)
  (definePiece 1 0 2 0)
  (definePiece 1 0 0 2)

  (definePiece 2 1 1 0)
  (definePiece 2 1 0 1)
  (definePiece 2 0 1 1)

  (definePiece 3 1 1 1)

  (set (vref pieceCount 0) 13.)
  (set (vref pieceCount 1) 3)
  (set (vref pieceCount 2) 1)
  (set (vref pieceCount 3) 1)
  (let ((m (fx+ 1 (fx* d (fx+ 1 d)))))
    (set kount 0)
      (if (not (fit 0 m))
          (format t "~%Error."))
      (if (not (and (trial (place 0 m)) (fx= kount 2005)))
          (format t "~%Failure.")))))

;;; Perm

(define-constant permrange 10)

(define permarray (make-vector permrange))
(lset pctr 0)
(declare local pctr)

(define-constant (swapp n m)
  (exchange (vref permarray n) (vref permarray m)))

(define (perm)
  (labels (


((initialize)
  (do ((i 1 (fx+ i 1)))
      ((fx> i 7) (return))
    (set (vref permarray i) (fx- i 1))))

((permute n)
  (set pctr (fx+ pctr 1))
  (cond ((fxN= n 1)
         (permute (fx- n 1))
         (do ((m (fx- n 1) (fx- m 1)))
             ((fx= m 0) (return))
           (swapp n m)
           (permute (fx- n 1))
           (swapp n m)))))

)
  (set pctr 0)
  (do ((i 0 (fx+ i 1)))
      ((fx>= i 5) (return))
    (initialize)
    (permute 7))
  (if (fxN= pctr 43300) (format t "~&Error in Perm. ~a ~%" pctr))
  (return)))

;;; Towers

(define peg-vector (make-vector 3))
(lset move-count 0)
(declare local move-count)

(define (towers)
  (labels (
  ((init ndiscs)
    (set (vref peg-vector 0) nil)
    (set (vref peg-vector 1) nil)
    (set (vref peg-vector 2) nil)
    (set move-count 0)
    (do ((disc-size ndiscs (fx- disc-size 1)))
        ((fx<= disc-size 0) t)
        (push-disc disc-size 0)))

  ((ltowers n i j)
    (cond ((fx= n 1)
           (move-disc i j))
          (else
           (let ((k (fx- (fx- 3 i) j)))
             (ltowers (fx- n 1) i k)
             (move-disc i j)
             (ltowers (fx- n 1) k j)))))

  ((move-disc peg-i peg-j)
    (push-disc (pop-disc peg-i) peg-j)
    (set move-count (fx+ move-count 1))
    (return))

  ((pop-disc i)
    (let ((disc (pop (vref peg-vector i))))
      (if (null? disc)
          (error "nothing to pop")
          disc)))

  ((push-disc disc i)
    (let ((top (car (vref peg-vector i))))
      (cond ((and (fixnum? top) (fx> disc top))
             (error "disc size error"))
            (else
             (push (vref peg-vector i) disc))))))

    (init 14)
    (ltowers 14 0 1)
    (if (fxN= move-count 16383)
        (error "in List-towers: move-count = ~a" move-count)
         move-count)))
```

```
;;; Intmm

(define-constant aref
  (object (lambda (array i j)
            (vref array (fx+ (fx* i rowsize) j)))
    ((setter self)
     (lambda (array i j val)
       (set (vref array (fx+ (fx* i rowsize) j)) val)))))

(define-constant rowsize 64)

(define ima (make-vector (* rowsize rowsize)))
(define imb (make-vector (* rowsize rowsize)))
(define imr (make-vector (* rowsize rowsize)))

(define (intmm)
  (labels (
((initmatrix matrix)
  (do ((i 0 (fx+ i 1)))
      ((fx>= i rowsize))
    (do ((j 0 (fx+ j 1)))
        ((fx>= j rowsize))
      (set (aref matrix i j) (fx* i j)))))

((innerproduct a b row column)
  (do ((i 0 (fx+ i 1))
       (sum 0 (fx+ sum (fx* (aref a row i) (aref b i column)))))
      ((fx>= i rowsize) sum))))

  (initmatrix ima)
  (initmatrix imb)
  (do ((i 0 (fx+ i 1)))
      ((fx>= i rowsize))
    (do ((j 0 (fx+ j 1)))
        ((fx>= j rowsize))
      (set (aref imr i j) (innerproduct ima imb i j))))))
```

Appendix C

Closure benchmarks

```
;;; Curry

(define (c3+ x)
  (lambda (y)
    (lambda (z)
      (fx+ (fx+ x y) z))))

(define (3+ x y z)
  (((c3+ x) y) z))


(define (fib3 x)
 (iterate fib3 ((x x))
  (cond ((fx<= x 3) 1)
        (else
         (3+ (fib3 (fx- x 1)) (fib3 (fx- x 2)) (fib3 (fx- x 3)))))))

(define (curry) (fib3 18))

;;; Kons

(define (kons x y)
  (lambda (key)
    (xcase key
      ((car) x)
      ((cdr) y))))

(define list-of-ones
  (do ((i 10000 (fx- i 1))
       (l '() (kons 1 l)))
      ((fx= i 0) l)))

(define (kar x) (x 'car))
(define (kdr x) (x 'cdr))

(define (sumlist)
  (do ((l list-of-ones (kdr l))
       (sum 0 (fx+ (kar l) sum)))
      ((null? l) sum)))
```

```
(define (sum)
  (+ (sumlist) (sumlist) (sumlist) (sumlist) (sumlist)))
```

Bibliography

[Abelson 85] Abelson, H. and Sussman G.J. with Sussman, J.
Structure and Interpretation of Computer Programs.
MIT Press, Cambridge, 1985.

[Aho 86] Aho, A. V., Sethi, R. and Ullman, J.D.
Compilers Principles, Techniques, and Tools.
Addison Wesley, 1986.

[Brooks 82] 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.

[Brooks 86] Brooks, R.A., Posner, D.B., McDonald, J.L., White, J.L., Benson, E. and Gabriel, R.P.
Design of an Optimizing, Dynamically Retargetable Compiler for Common Lisp.
In *Conference Record of the 1986 ACM Symposium on Lisp and Functional Programming,* pages 67-85.

[Chaiten 81] Chaiten, G.J. et al.
Register Allocation via Coloring.
Computer Languages, Vol. 6, pp. 45-47, 1981, Great Britain.

[Chaiten 82] Chaiten, G.J
Register Allocation and Spilling via Graph Coloring.
SIGPLAN Symp. on Compiler Construction, June 23-25, 1982, Boston, Mass.

[Chow 84] Chow, F. and Hennessy, J.
Register Allocation by Priority-based Coloring.
In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 222–232. ACM, June 1984.

[Clinger 85] Clinger, W. editor.
The revised revised report on Scheme, or an uncommon Lisp.
MIT Artificial Intelligence Memo 848, August 1985.
Also published as Computer Science Department Technical Report 174, Indiana University, June 1985.

[Ellis 85] Ellis, J.R.
Bulldog: A Compiler for VLIW Architectures.
Ph.D. Th., Yale University., 1985. available as Research Report YALEU/DCS/RR-364.

[Felleisen 87] Felleisen, M., Friedman, D.P., Kohlbecker, E. and Duba, B.
A Syntactic Theory on Sequential Control.
Tech. Report. 215, Indiana University, Feb., 1987.

[Fisher 81] Fisher, J.A.
Trace Scheduling: A technique for global microcode compaction.
IEEE Transactions on Computers C-30(7):478-490, July 1981.

[Gabriel 85] Gabriel, R.P.
Performance and Evaluation of Lisp Systems.
MIT Press, Cambridge, Mass., 1985.

[Griss 82] Griss, M.L. and Benson, E.
Current Status of a Portable Lisp Compiler.
SIGPLAN Symp. on Compiler Construction, June 23-25, 1982, Boston, Mass.

[Haynes 84] Haynes, C.T., Friedman, D.P. and Wand, M.
Continuations and coroutines.
In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming,* pages 293–298.

[Haynes 84b] Haynes, C.T. and Friedman, D.P.
Engines build process abstracions.
In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming,* pages 18–24.

[Hudak 84] Hudak, P.
ALFL Reference Manual and Programmers Guide.
Dept of Computer Science, University of Yale, Technical Report YALEU/DCS/TR-322 Second Edition October 1984

[Hudak 84b] Hudak P. and Kranz, D.A.
A combinator-based compiler for a functional language.
Conference Record of the Eleventh Annual ACM Symposium on Principles of Programming Languages.
January, 1984.

[Karr 84] Karr, M.
Code Generation by Coagulation.
In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, pages 1–12. ACM, June 1984.

[Kelsey] Kelsey, R.
Ph.D. dissertation expected June 1988.

[Kranz 86] Kranz D.A., Kelsey, R., Rees J.A., Hudak P., Philbin, J. and Adams, N.I.
Orbit: An optimizing compiler for Scheme.
In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 219–233. ACM, June 1986.

[Mycroft 81] Mycroft, A.
Abstract Interpretation and Optimizing Transformations for Applicative Programs.
Ph.D. Th., Univ. of Edinburgh, 1981.

[Powell 84] 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.

[Rees 82] Rees, J.A. and Adams, N.I.
T: A dialect of Lisp or, lambda: The ultimate software tool.
In *Conference Record of the 1982 ACM Symposium on Lisp and Functional Programming*, pages 114–122.

[Rees 84] Rees, J.A., Adams, N.I. and Meehan, J.R.
The T manual, fourth edition.
Yale University Computer Science Department, January 1984.

[Rees] Rees, J.A. (personal communication)

[Rozas] Rozas, W. (personal communication).

[Slade 87] Slade, S.
The T Programming Language.
Prentice-Hall, Inc. 1987.

[Steele 76] Steele, G.L. Jr. and Sussman, G.J.
Lambda, the ultimate imperative.
MIT Artificial Intelligence Memo 353, March 1976.

[Steele 76b] Steele, G.L. Jr.
Lambda, the ultimate declarative.
MIT Artificial Intelligence Memo 379, November 1976.

[Steele 77] Steele, G.L. Jr.
Debunking the "expensive procedure call" myth, or procedure call implementations considered harmful, or lambda, the ultimate GOTO.
In *ACM Conference Proceedings*, pages 153–162. ACM, 1977.

[Steele 78] Steele, G.L. Jr. and Sussman, G.J.
The revised report on Scheme, a dialect of Lisp.
MIT Artificial Intelligence Memo 452, January 1978.

[Steele 78b] Steele, G.L. Jr.
Rabbit: a compiler for Scheme.
MIT Artificial Intelligence Laboratory Technical Report 474, May 1978.

[Steele 84] Steele, G.L. Jr. *Common Lisp: The Language.* Digital Press, Burlington MA, 1984.

[Sun 86] Sun Common Lisp Performance report.
Sun Microsystems, Inc.
July 11, 1986.

[Szymanski 78] Szymanski.
Assembling code for machines with span-dependent instructions.
CACM 21, 4 (April 1978), pp. 300-308.

[Wand 80] Wand, M.
Continuation-based multiprocessing.
In *Conference Record of the 1980 Lisp Conference*, pages 19–28.
The Lisp Conference, August 1980.

[Wulf 75] Wulf, W., Johnson, R., Weinstock, C., Hobbs, S. and Geschke, C.
The Design of an Optimizing Compiler.
American Elsevier, 1975.