COMPILER OPTIMIZATION BASED ON VIEWING LAMBDA AS RENAME PLUS GOTO

GUY L. STEELE, JR.

In this section, Guy Steele describes a compiler for the lexically-scoped dialect of LISP known as SCHEME. The compiler knows relatively little about specific data manipulation primitives such as arithmetic operators, but concentrates on general issues of environment and control. Rather than having specialized knowledge about a large variety of control and environment constructs, the compiler handles only a small basis set that reflects the semantics of lambda calculus. All of the traditional imperative constructs, such as sequencing, assignment, looping, and GOTO, as well as many standard LISP constructs such as AND, OR, and COND, are expressed as macros in terms of the applicative basis set. A small number of optimization techniques, coupled with the treatment of function calls as GOTO statements, serves to produce code as good as that produced by more traditional compilers. The macro approach enables speedy implementation of new constructs without sacrificing efficiency in the generated code.

Lexically-scoped LISP

We have developed a compiler for the lexically-scoped dialect of LISP known as SCHEME. The compiler knows relatively little about specific data manipulation primitives such as arithmetic operators, but concentrates on general issues of environment and control. Rather than having specialized knowledge about a large variety of control and environment constructs, the compiler handles only a small basis set which reflects the semantics of lambda-calculus. All of the traditional imperative constructs, such as sequencing, assignment, looping, GOTO, as well as many standard LISP constructs such as AND, OR, and COND, are expressed as macros in terms of the applicative basis set. A small number of optimization techniques, coupled with the treatment of function calls as GOTO statements, serve to produce code as good as that produced by more traditional compilers. The macro approach enables speedy implementation of new constructs as desired without sacrificing efficiency in the generated code.

A subset of SCHEME serves as the representation intermediate between the optimized SCHEME code and the final output code; code is expressed in this subset in the so-called continuation-passing style. As a subset of SCHEME, it enjoys the same theoretical properties; one could even apply the same optimizer used on the input code to the intermediate code. However, the subset is so chosen that all temporary quantities are made manifest as variables, and no control stack is needed to evaluate it. As a result, this apparently applicative representation admits an imperative interpretation which permits easy transcription to final imperative machine code.

Background

In 1975 Sussman and Steele reported on the implementation of SCHEME, a dialect of LISP with the properties of lexical scoping and tail-recursion; this implementation is embedded

within MacLISP [Moon 1974], a version of LISP which does not have these properties. The property of lexical scoping (that a variable can be referenced only from points textually within the expression which binds it) is a consequence of the fact that all functions are closed in the "binding environment." [Moses 1970] That is, SCHEME is a "full-funarg" LISP dialect. The property of tail-recursion implies that loops written in an apparently recursive form will actually be executed in an iterative fashion. Intuitively, function calls do not "push control stack;" instead, it is argument evaluation which pushes control stack. The two properties of lexical scoping and tail-recursion are not independent. In most LISP systems [McCarthy 1962] [Moon 1974] [Teitelman 1975], which use dynamic scoping rather than lexical, tail-recursion is impossible because function calls must push control stack in order to be able to undo the dynamic bindings after the return of the function. On the other hand, it is possible to have a lexically scoped LISP which does not tail-recurse, but it is easily seen that such an implementation only wastes storage space needlessly compared to a tail-recursing implementation. Together, these two properties cause SCHEME to reflect lambda-calculus semantics much more closely than dynamically scoped LISP systems. SCHEME also permits the treatments of functions as full-fledged data objects; they may be passed as arguments, returned as values, made part of composite data structures, and notated as independent, unnamed ("anonymous") entities. (Compare this with most ALGOL-like languages, in which a function can be written only by declaring it and giving it a name; then imagine being able to use an integer value only by giving it a name in a declaration!) The property of lexical scoping allows this to be done in a consistent manner without the possibility of identifier conflicts (that is, SCHEME "solves the FUNARG problem"). Also of interest is the technique of "continuation-passing style," a way of writing programs in SCHEME such that no function ever returns a value.

Since the first appearance of SCHEME we have continued

to explore the properties of lexical scoping and tail-recursion, and various coding styles. These properties and styles of coding may be exploited to implement most traditional programming constructs, such as assignment, looping, and call-by-name, in terms of function application. Such applicative (lambda-calculus) models of programming language constructs are well-known to theoreticians (see [Stoy 1974], for example), but have not been used in a practical programming system. All of these constructs are actually made available in SCHEME by macros which expand into these applicative definitions. This technique has permitted the speedy implementation of a rich user-level language in terms of a very small, easy-to-implement basis set of primitive constructs. The escape operator CATCH is easily modelled by transforming a program into the continuation-passing style. Transforming a program into this style enforces a particular order of argument evaluation, and makes all intermediate computational quantities manifest as variables.

Examination of the properties of tail-recursion shows that the usual view of function calls as pushing a return address must lead to an either inefficient or inconsistent implementation, while the tail-recursive approach of SCHEME leads to a uniform discipline in which function calls are treated as GOTO statements which also pass arguments. A consequence of lexical scoping is that the only code which can reference the value of a variable is in a given environment is code which is closed in that environment or which received the value as an argument; this in turn implies that a compiler can structure a run-time environment in any arbitrary fashion, because it will compile all the code which can reference that environment, and so can arrange for that code to reference it in the appropriate manner. Such references do not require any kind of search (as is commonly and incorrectly believed in the LISP community because of early experience with LISP interpreters which search a-lists) because the compiler can determine the precise location of each variable in an environment at compile time. It is not necessary to use a standard format, because neither interpreted

code nor other compiled code can refer to that environment.

Transforming a program into the continuation-passing style elucidates traditional compilation issues such as register allocation because user variables and intermediate quantities alike are made manifest as variables on an equal footing. We have exhibited an algorithm for converting any SCHEME program (not containing ASET) to continuation-passing style.

We have implemented two compilers for the language SCHEME. The purpose was to explore compilation techniques for a language modelled on lambda-calculus, using lambda-calculus-style models of imperative programming constructs. Both compilers use the strategy of converting the source program to continuation-passing style.

The first compiler was a throw-away version which, while usable, was written only to clear up some technical points of implementation. The second compiler, with which we are primarily concerned here, is known as RABBIT. It, like the first, is written almost entirely in SCHEME (with minor exceptions due only to problems in interfacing with certain MacLISP I/O facilities). However, it is much more clever. It is intended to demonstrate a number of optimization techniques related to lexical environment and tail-recursive control structures.

The Thesis

- Function calls are not expensive when compiled correctly; they should be thought of as GOTO statements that pass arguments.

- The combination of cheap function calls, lexical scoping, tail-recursion, and anonymous notation of functions permits the definition of a wide variety of "imperative" constructs in applicative terms. Because these properties result from adhering to the principles of the well-known lambda-calculus, such definitions can be lifted intact from

existing literature and used directly.

- A macro facility (the ability to specify syntactic transformations) makes it practical use these as the only definitions of imperative constructs in a programming system. Such a facility makes it extremely easy to define new constructs.

- A few well-chosen optimization strategies enable the compilation of these applicative definitions into the imperative low-level code which one would expect from a traditional compiler.

- The macro facility and the optimization techniques used by the compiler can be conceptually unified. The same properties which make it easy to write the macros make it easy to define optimizations correctly. Just as many programming constructs are defined in terms of a small, well-chosen basis set, so a large number of traditional optimization techniques fall out as special cases of the few used in RABBIT. This is no accident. The separate treatment of a large and diverse set of constructs necessitates separate optimization techniques for each. As the basis set of constructs is reduced, so is the set of interesting transformations.

- The technique of compiling by converting to continuation-passing style elucidates some important compilation issues in a natural way. Intermediate quantities are made manifest; so is the precise order of evaluation. Moreover, this is all expressed in a language isomorphic to a subset of the source language SCHEME; as a result the continuation-passing style version of a program inherits many of the phisophical and practical advantages. For example, the same optimization techniques can be applied at this level as at the original

source level. While the use of continuation-passing style may not make the decisions any easier, it provides an effective and natural way to express the results of those decisions.

- Continuation-passing style, while apparently applicative in nature, admits a peculiarly imperative interpretation as a consequence of the facts that it requires no control stack to be evaluated and that no functions ever return values. As a result, it is easily converted to an imperative machine language.

- A SCHEME compiler should ideally be a designer of good data structures, since it may choose any representation whatsoever for environments. RABBIT has a rudimentary design knowledge, involving primarily the preferral of registers to heap-allocated storage. However, there is room for knowledge of "bit-diddling" representations.

- We suggest that those who have tried to design useful UNCOL's (UNiversal Computer-Oriented Languages) [Sammet 1969] [Coleman 1974] have perhaps been thinking too imperatively, and worrying more about data manipulation primitives than about environment and control issues. As a result, proposed UNCOLs have been little more than generalizations of contemporary machine languages. We suggest that SCHEME makes an ideal UNCOL at two levels. The first level is the fully applicative level, to which most source-language constructs are easily reduced; the second is the continuation-passing style level, which is easily reduced to machine language. We envision building a compiler in three stages: reduction of a user language to basic SCHEME, whether by macros, a parser of algebraic syntax, or some other means; transformation to continuation-passing style; and

generation of code for a particular machine. RABBIT addresses itself to the middle stage. Data manipulation primitives are completely ignored at this stage, and are just passed along from input to output. These primitives, whether integer arithmetic, string concatenation and parsing, or list structure manipulators, are chosen as a function of a particular source language and a particular target machine. RABBIT deals only with fundamental environment and control issues common to any mode of algorithmic expression.

- While most syntactic issues tend to be rather superficial, we point out that algebraic syntax tends to obscure the fundamental nature of function calling and tail-recursion by arbitrarily dividing functions into syntactic classes such as "operators" and "functions." ([Standish 1976], for example, uses much space to exhibit each conceptually singular transformation in a multiplicity of syntactic manifestations.) The lack of an "anonymous" notation for functions in most algebraic languages, and the inability to treat functions as data objects, is a distinct disadvantage. The uniformity of LISP syntax makes these issues easier to deal with.

To the LISP community in particular we address these additional points:

- Lexical scoping need not be as expensive as is commonly thought. Experience with lexically-scoped *interpreters* is misleading. Lexical scoping is not inherently slower than dynamic scoping. While some implementations may entail access through multiple levels of structure, this occurs only under circumstances (accessing of variables through multiple levels of closure) which could not even be expressed in a dynamically scoped language! Unlike deep-bound dynamic variables, lexical access requires no

search; unlike shallow-bound dynamic variables, lexical binding does not require that values be put in a canonical value cell. The compiler has complete discretion over the manipulation of environments and variable values.

- Lexical scoping does not necessarily make LISP programming unduly difficult. The very existence of RABBIT, a working compiler some fifty pages in length written in SCHEME, implemented in about a month, part-time, substantiates this claim (which is, however, admitted to be mostly a matter of taste and experience). SCHEME has also been used to implement several AI problem-solving languages, including AMORD [Doyle 1977].

The Source Language - SCHEME

The basic language processed by RABBIT is a subset of the SCHEME language as described in [Sussman 1975], the primary restrictions being that the first argument to ASET must be quoted and that the multiprocessing primitives are not permitted. This subset is summarized here.

SCHEME is essentially a lexically scoped ("full funarg") dialect of LISP. Interpreted programs are represented by S-expressions in the usual manner. Numbers represent themselves. Atomic symbols are used as identifiers (with the conventional exception of T and NIL, which are conceptually treated as constants). All other constructs are represented as lists.

In order to distinguish the various other constructs, SCHEME follows the usual convention that a list whose car is one of a set of distinguished atomic symbols is treated as directed by a rule conceptually associated with that symbol. All other lists (those with non-atomic cars, or with undistinguished atoms in their cars) are combinations, or function calls. All subforms of the list are uniformly evaluated in an unspecified

order, and then the value of the first (the function) is applied to the values of all the others (the arguments). Notice that the function position is evaluated in the same way as the argument positions (unlike most other LISP systems).

The atomic symbols which distinguish special constructs are as follows:

LAMBDA This denotes a function. A form (LAMBDA (var1 var2 ... varn) body) will evaluate to a function of n arguments. The parameters vari are identifiers (atomic symbols) which may be used in the body to refer to the respective arguments when the function is invoked. Note that a LAMBDA-expression is not a function, but evaluates to one, a crucial distinction.

IF This denotes a conditional form. (IF a b c) evaluates the predicate a, producing a value x; if x is non-NIL, then the consequent b is evaluated, and otherwise the alternative c. If c is omitted, NIL is assumed.

QUOTE As in all LISP systems, this provides a way to specify any S-expression as a constant. (QUOTE x) evaluates to the S-expression x. This may be abbreviated to 'x, thanks to the MacLISP read-macro-character feature.

LABELS This primitive permits the local definition of one or more mutually recursive functions. The format is:

```
(LABELS ((name1 (LAMBDA ...))
         (name2 (LAMBDA ...))
         ...
         (namen (LAMBDA ...)))
        body)
```

This evaluates the body in an environment in which the names refer to the respective functions, which are themselves closed in that same environment. Thus references to these names in the bodies of the LAMBDA-expressions will refer to the labelled

functions.

ASET This is the primitive side-effect on variables. (ASET' var body) evaluates the body, assigns the resulting value to the variable var, and returns that value. Note the use of "'" to quote the variable name. The SCHEME interpreter permits one to compute the name of the variable, but for technical and philosophical reasons RABBIT forbids this.

CATCH This provides an escape operator facility. [Landin 1965] [Reynolds 1972] (CATCH var body) evaluates the body, which may refer to the variable var, which will denote an "escape function" of one argument which, when called, will return from the CATCH-form with the given argument as the value of the CATCH-form.

Macros Any atomic symbol which has been defined in one of various ways to be a macro distinguishes a special construct whose meaning is determined by a macro function. This function has the responsibility of rewriting the form and returning a new form to be evaluated in place of the old one. In this way complex syntactic constructs can be expressed in terms of simpler ones.

The Target Language

The "target language" is a highly restricted subset of MacLISP, rather than any particular machine language for an actual hardware machine such as the PDP-10. RABBIT produces MacLISP function definitions which are then compiled by the standard MacLISP compiler. In this way we do not need to deal with the uninteresting vagaries of a particular piece of hardware, nor with the peculiarities of the many and various data-manipulation primitives (CAR, RPLACA, +, etc.). We allow the MacLISP compiler to deal with them, and concentrate on the issues of environment and control which are unique to SCHEME. While for production use this is mildly inconvenient (since the

code must be passed through two compilers before use), for research purposes it has saved the wasteful re-implementation of much knowledge already contained in the MacLISP compiler.

On the other hand, the use of MacLISP as a target language does not by any means trivialize the task of RABBIT. The MacLISP function-calling mechanism cannot be used as a target construct for the SCHEME function call, because MacLISP's function calls are not guaranteed to behave tail-recursively. Since tail-recursion is a most crucial characteristic distinguishing SCHEME from most LISP systems, we must implement SCHEME function calls by more primitive methods. Similarly, since SCHEME is a full-funarg dialect of LISP while MacLISP is not, we cannot in general use MacLISP's variable-binding mechanisms to implement those of SCHEME. On the other hand, it is a perfectly legitimate optimization to use MacLISP mechanisms in those limited situations where they are applicable.

Language Design Considerations

We divide the definition of the SCHEME language into two parts: the environment and control constructs, and the data manipulation primitives. Examples of the former are LAMBDA-expressions, combinations, and IF; examples of the latter are CONS, CAR, EQ, and PLUS. Note that we can conceive of a version of SCHEME which did not have CONS, for example, and more generally did not have S-expressions in its data domain. Such a version would still have the same environment and control constructs, and so would hold the same theoretical interest for our purposes here. (Such a version, however, would be less convenient for purposes of writing a meta-circular description of the language, however!)

SCHEME is an applicative language which conforms to the essential properties of the axioms obeyed by lambda-calculus [Church 1965] expressions. Among these are the rules of alpha-conversion and beta-conversion. The first intuitively

implies that we can uniformly rename a function parameter and all references to it without altering the meaning of the function. An important corollary to this is that we can effectively locate all the references. The second implies that in a situation where a known function is being called with known argument expressions, we may substitute an argument expression for a parameter reference within the body of the function (provided no naming conflicts result). Both of these operations are of importance to an optimizing compiler. Another property which follows indirectly is that of tail-recursion. This property is exploited in expressing iteration in terms of applicative constructs.

There are those to whom lexical scoping is nothing new, for example the ALGOL community. For this audience, however, we should draw attention to another important feature of SCHEME, which is that functions are first-class data objects. They may be assigned or bound to variables, returned as values of other functions, placed in arrays, and in general treated as any other data object. Just as numbers have certain operations defined on them, such as addition, so functions have an important operation defined on them, namely invocation.

The ability to treat functions as objects is not at all the same as the ability to treat *representations* of functions as objects. It is the latter ability that is traditionally associated with LISP; functions can be represented as S-expressions. In a version of SCHEME which had no S-expression primitives, however, one could still deal with functions (i.e. closures) as such, for that ability is part of the fundamental environment and control facilities. Conversely, in a SCHEME which does have CONS, CAR, and CDR, there is no defined way to use CONS by itself to construct a function (although a primitive ENCLOSE is now provided which converts an S-expression representation of a function into a function), and the CAR or CDR of a function is in general undefined. The only defined operation on a function is invocation.

We draw this sharp distinction between environment and control constructs on the one hand and data manipulation

primitives on the other because only the former are treated in any depth by RABBIT, whereas much of the knowledge of a "real" compiler deals with the latter. A PL/I compiler must have much specific knowledge about numbers, arrays, strings, and so on. We have no new ideas to present here on such issues, and so have avoided this entire area. RABBIT itself knows nothing about data manipulation primitives beyond being able to recognize them and pass them along to the output code, which is a small subset of MacLISP. In this way RABBIT can concentrate on the interesting issues of environment and control, and exploit the expert knowledge of data manipulation primitives already built into the MacLISP compiler.

The Use of Macros

An important characteristic of the SCHEME language is that its set of primitive constructs is quite small. This set is not always convenient for expressing programs, however, and so a macro facility is provided for extending the expressive power of the language. A macro is best thought of as a syntax rewrite rule. As a simple example, suppose we have a primitive GCD which takes only two arguments, and we wish to be able to write an invocation of a GCD function with any number of arguments. We might then define (in a production-rule style) the conditional rule:

```
(XGCD)          => 0
(XGCD x)        => x
(XGCD x . rest) => (GCD x (XGCD . rest))
```

(Notice the use of LISP dots to refer to the rest of a list.) This is not considered to be a definition of a function XGCD, but a purely syntactic transformation. In principle all such transformations could be performed before executing the program. In fact, RABBIT does exactly this, although the SCHEME interpreter naturally does it incrementally, as each

macro call is encountered.

Rather than use a separate production rule and pattern-matching language, in practice SCHEME macros are defined as transformation functions from macro-call expressions to resulting S-expressions, just as they are in MacLISP. (Here, however, we shall continue to use production rules for purposes of exposition.) It is important to note that macros need not be written in the language for which they express rewrite rules; rather, they should be considered an adjunct to the interpreter, and written in the same language as the interpreter (or the compiler). To see this more clearly, consider again a version of SCHEME which does not have S-expressions in its data domain. If programs in this language are represented as S-expressions, then the interpreter for that language cannot be written in that language, but in another meta-language which does deal with S-expressions. Macros, which transform one S-expression (representing a macro call) to another (the replacement form, or the interpretation of the call), clearly should be expressed in this meta-language also. The fact that in most LISP systems the language and the meta-language appear to coincide is a source of both power and confusion.

Let us consider some typical macros used in SCHEME. The BLOCK macro is similar to the MacLISP PROGN; it evaluates all its arguments and returns the value of the last one. One critical characteristic is that the last argument is evaluated "tail-recursively" (quotation marks are used because normally we speak of invocation, not evaluation, as being tail-recursive). An expansion rule is given for this in [Steele 1976A] equivalent to:

```
(BLOCK x)          =>  x
(BLOCK x . rest)   =>  ((LAMBDA (DUMMY) (BLOCK . rest)) x)
```

This definition exploits the fact that SCHEME is evaluated in applicative order, and so will evaluate all arguments before applying a function to them. Thus, in the second subrule, x must be evaluated, and then the block of all the rest is. It is

then clear from the first subrule that the last argument is evaluated "tail-recursively."

One problem with this definition is the occurrence of the variable DUMMY, which must be chosen so as not to conflict with any variable used by the user. This we refer to as the "GENSYM problem," in honor of the traditional LISP function which creates a "fresh" symbol. It would be nicer to write the macro in such a way that no conflict could arise no matter what names were used by the user. There is indeed a way, which ALGOL programmers will recognize as equivalent to the use of "thunks," or call-by-name parameters:

```
(BLOCK x)          =>  x
(BLOCK x . rest)   =>  ((LAMBDA (A B) (B))
                        x
                        (LAMBDA () (BLOCK . rest)))
```

This is a technique which should be understood quite thoroughly, since it is the key to writing correct macro rules without any possibility of conflicts between names used by the user and those needed by the macro. As another example, let us consider the AND and OR constructs as used by most LISP systems. OR evaluates its arguments one by one, in order, returning the first non-NIL value obtained (without evaluating any of the following arguments), or NIL if all arguments produce NIL. AND is the dual to this; it returns NIL if any argument does, and otherwise the value of the last argument. A simple-minded approach to OR would be:

```
(OR)           =>  'NIL
(OR x . rest)  =>  (IF x x (OR . rest))
```

There is an objection to this, which is that the code for x is duplicated. Not only does this consume extra space, but it can execute erroneously if x has any side-effects. We must arrange to evaluate x only once, and then test its value:

```
(OR)          =>  'NIL
(OR x . rest) =>  ((LAMBDA (V) (IF V V (OR . rest))) x)
```

This certainly evaluates x only once, but admits a possible naming conflict between the variable V and any variables used by rest. This is avoided by the same technique used for BLOCK:

```
(OR)          =>  'NIL
(OR x . rest) =>  ((LAMBDA (V R) (IF V V (R)))
                    x
                    (LAMBDA () (OR . rest)))
```

Let us now consider a rule for the more complicated COND construct:

```
(COND)  =>  'NIL
(COND (x) . rest)  =>  (OR x (COND . rest))
(COND (x . r) . rest)  =>  (IF x (BLOCK . r) (COND . rest))
```

This defines the "extended" COND of modern LISP systems, which produces NIL if no clauses succeed, which returns the value of the predicate in the case of a singleton clause, and which allows more than one consequent in a clause. An important point here is that one can write these rules in terms of other macro constructs such as OR and BLOCK.

SCHEME also provides macros for such constructs as DO and PROG, all of which expand into similar kinds of code using LAMBDA, IF, and LABELS (see below). In particular, PROG permits the use of GO and RETURN in the usual manner. In this way all the traditional imperative constructs are expressed in an applicative style.

None of this is particularly new; theoreticians have modelled imperative constructs in these terms for years. What is new, we think, is the serious proposal that a practical interpreter and compiler can be designed for a language in which such

models serve as the sole definitions of these imperative constructs. This approach has both advantages and disadvantages.

One advantage is that the base language is small. A simple-minded interpreter or compiler can be written in a few hours. (We have re-implemented the SCHEME interpreter from scratch a half-dozen times or more to test various representation strategies; this was practical only because of the small size of the language.) Once the basic interpreter is written, the macro definitions for all the complex constructs can be used without revision. Moreover, the same macro definitions can be used by both interpreter and compiler (or by several versions of interpreter and compiler!). It is not necessary to "implement a construct twice," once each in interpreter and compiler.

Another advantage is that new macros are very easy to write (using facilities provided in SCHEME). One could easily invent a new kind of DO loop, for example, and implement it in SCHEME for both interpreter and all compilers in less than five minutes.

A third advantage is that the attention of the compiler can be focused on the basic constructs. Rather than having specialized code for two dozen different constructs, the compiler can have much deeper knowledge about each of a few basic constructs. One might object that this "deeper knowledge" consists of recognizing the two dozen special cases represented by the separate constructs of the former case. This is true to some extent. It is also true, however, that in the latter case such deep knowledge will carry over to any new constructs which are invented and represented as macros.

Among the disadvantages of the macro approach are lack of speed and the discarding of information. Many people have objected that macros are of necessity slower than, say, the FSUBR implementation used by most LISP systems. This is true in many current interpretive implementations, but need not be true of compilers or more cleverly designed interpreters. Moreover, the FSUBR implementation is not general; it is very hard for a user to write a meaningful FSUBR and then describe

to the compiler the best way to compile it. The macro approach handles this difficulty automatically. We do not object to the use of the FSUBR mechanism as a special-case "speed hack" to improve the performance of an interpreter, but we insist on recognizing the fact that it is not as generally useful as the macro approach.

Another objection relating to speed is that the macros produce convoluted code involving the temporary creation and subsequent invocation of many closures. We feel, first of all, that the macro writer should concern himself more with producing correct code than fast code. Furthermore, convolutedness can be eliminated by a few simple optimization techniques in the compiler, to be discussed below. Finally, function calls need not be as expensive as is popularly supposed. [Steele 1977A]

Information is discarded by macros in the situation, for example, where a DO macro expands into a large mess that is not obviously a simple loop; later compiler analysis must recover this information. This is indeed a problem. We feel that the compiler is probably better off having to recover the information anyway, since a deep analysis allows it to catch other loops which the user did not use DO to express for one reason or another. Another is the possibility that DO could leave clues around in the form of declarations if desired.

The Imperative Treatment of Applicative Constructs

Given the characteristics of lexical scoping and tail-recursive invocations, it is possible to assign a peculiarly imperative interpretation to the applicative constructs of SCHEME, which consists primarily of treating a function call as a GOTO. More generally, a function call is a GOTO that can pass one or more items to its target; the special case of passing no arguments is precisely a GOTO. It is never necessary for a function call to save a return address of any kind. It is true that return addresses are generated, but we adopt one of two other points of

view, depending on context. One is that the return address, plus any other data needed to carry on the computation after the called function has returned (such as previously computed intermediate values and other return addresses) are considered to be packaged up into an additional argument (the continuation) which is passed to the target. This lends itself to a non-functional interpretation of LAMBDA, and a method of expressing programs called the continuation-passing style, to be discussed further below. The other view, more intuitive in terms of the traditional stack implementation, is that the return address should be pushed before evaluating arguments rather than before calling a function. This view leads to a more uniform function-calling discipline, and is discussed in [Steele 1976B].

We are led by this point of view to consider a compilation strategy in which function calling is to be considered very cheap (unlike the situation with PL/I and ALGOL, where programmers avoid procedure calls like the plague -- see [Steele 1977A] for a discussion of this). In this light the code produced by the sample macros above does not seem inefficient, or even particularly convoluted. Consider the expansion of (OR a b c):

```
((LAMBDA (V R) (IF V V (R)))
 a
 (LAMBDA () ((LAMBDA (V R) (IF V V (R)))
             b
             (LAMBDA () ((LAMBDA (V R) (IF V V (R)))
                         c
                         (LAMBDA () 'NIL))))))
```

Then we might imagine the following (slightly contrived) compilation scenario. First, for expository purposes, we shall rename the variables in order to be able to distinguish them.

```
((LAMBDA (V1 R1) (IF V1 V1 (R1)))
 a
 (LAMBDA () ((LAMBDA (V2 R2) (IF V2 V2 (R2)))
             b
             (LAMBDA () ((LAMBDA (V3 R3) (IF V3 V3 (R3)))
                         c
                         (LAMBDA () 'NIL))))))
```

We shall assign a generated name to each LAMBDA-expression, which we shall notate by writing the name after the word LAMBDA. These names will be used as tags in the output code.

```
((LAMBDA name1 (V1 R1) (IF V1 V1 (R1)))
 a
 (LAMBDA name2 () ((LAMBDA name3 (V2 R2) (IF V2 V2 (R2)))
             b
             (LAMBDA name4 () ((LAMBDA name5 (V3 R3)
                                     (IF V3 V3 (R3)))
                         c
                         (LAMBDA name6 () 'NIL))))))
```

Next, a simple analysis shows that the variables R1, R2, and R3 always denote the LAMBDA-expressions named name2, name4, and name6, respectively. Now an optimizer might simply have substituted these values into the bodies of name1, name3, and name5 using the rule of beta-conversion, but we shall not apply that technique here. Instead we shall compile the six functions in a straightforward manner. We make use of the additional fact that all six functions are closed in identical environments (we count two environments as identical if they involve the same variable bindings, regardless of the number of "frames" involved; that is, the environment is the same inside and outside a (LAMBDA () ...)). Assume a simple target machine with argument registers called reg1, reg2, etc.

```
main:   <code for a>            ;result in reg1
        LOAD reg2,[name2]       ;[name2] is the closure for name2
        CALL-FUNCTION 2,[name1] ;call name1 with 2 arguments

name1:  JUMP-IF-NIL reg1,name1a
        RETURN                  ;return the value in reg1
name1a: CALL-FUNCTION 0,reg2    ;call function in reg2, 0 arguments

name2:  <code for b>            ;result in reg1
        LOAD reg2,[name4]       ;[name4] is the closure for name4
        CALL-FUNCTION 2,[name3] ;call name3 with 2 arguments

name3:  JUMP-IF-NIL reg1,name3a
        RETURN                  ;return the value in reg1
name3a: CALL-FUNCTION 0,reg2    ;call function in reg2, 0 arguments

name4:  <code for c>            ;result in reg1
        LOAD reg2,[name6]       ;[name6] is the closure for name6
        CALL-FUNCTION 2,[name5] ;call name5 with 2 arguments

name5:  JUMP-IF-NIL reg1,name5a
        RETURN                  ;return the value in reg1
name5a: CALL-FUNCTION 0,reg2    ;call function in reg2, 0 arguments

name6:  LOAD reg1,'NIL          ;constant NIL in reg1
        RETURN
```

Now we make use of our knowledge that certain variables always denote certain functions, and convert CALL-FUNCTION of a known function to a simple GOTO. (We have actually done things backwards here; in practice this knowledge is used before generating any code. We have fudged over this issue here, but will return to it later. Our purpose here is merely to demonstrate the treatment of function calls as GOTOs.)

```
main:   <code for a>            ;result in reg1
        LOAD reg2,[name2]       ;[name2] is the closure for name2
        GOTO name1

name1:  JUMP-IF-NIL reg1,name1a
        RETURN                  ;return the value in reg1
name1a: GOTO name2

name2:  <code for b>            ;result in reg1
        LOAD reg2,[name4]       ;[name4] is the closure for name4
        GOTO name3

name3:  JUMP-IF-NIL reg1,name3a
        RETURN                  ;return the value in reg1
name3a: GOTO name4

name4:  <code for c>            ;result in reg1
        LOAD reg2,[name6]       ;[name6] is the closure for name6
        GOTO name5

name5:  JUMP-IF-NIL reg1,name5a
        RETURN                  ;return the value in reg1
name5a: GOTO name6

name6:  LOAD reg1,'NIL          ;constant NIL in reg1
        RETURN
```

The construction indicates the creation of a closure for foo in the current environment. This will actually require additional instructions, but we shall ignore the mechanics of this for now since analysis will remove the need for the construction in this case. The fact that the only references to the variables R1, R2, and R3 are function calls can be detected and the unnecessary LOAD instructions eliminated. (Once again, this would actually be determined ahead of time, and no LOAD instructions would be generated in the first place. All of this is determined by a

general pre-analysis, rather than a peephole post-pass.) Moreover, a GOTO to a tag which immediately follows the GOTO can be eliminated.

```
main:   <code for a>            ;result in regl
name1:  JUMP-IF-NIL regl,name1a
        RETURN                  ;return the value in regl
name1a:
name2:  <code for b>            ;result in regl
name3:  JUMP-IF-NIL regl,name3a
        RETURN                  ;return the value in regl
name3a:
name4:  <code for c>            ;result in regl
name5:  JUMP-IF-NIL regl,name5a
        RETURN                  ;return the value in regl
name5a:
name6:  LOAD regl,'NIL          ;constant NIL in regl
        RETURN
```

This code is in fact about what one would expect out of an ordinary LISP compiler. (There is admittedly room for a little more improvement.) RABBIT indeed produces code of essentially this form, by the method of analysis outlined here.

Similar considerations hold for the BLOCK macro. Consider the expression (BLOCK a b c); conceptually this should perform a, b, and c sequentially. In fact, the code produced by the same methods used for OR above is:

```
main:   <code for a>
name1:
name2:  <code for b>
name3:
name4:  <code for c>
        RETURN
```

which is precisely what is desired.

Notice that this has fallen out of a general strategy involving only an approach to compiling function calls, and has involved no special knowledge of OR or BLOCK not encoded in the macro rules. The cases shown so far are actually special cases of a more general approach, special in that all the conceptual closures involved are closed in the same environment, and called from places that have not disturbed that environment, but only used "registers." In the more general case, the environments of caller and called function will be different. This divides into two subcases, corresponding to whether the closure was created by a simple LAMBDA or by a LABELS construction. The latter involves circular references, and so is somewhat more complicated; but it is easy to show that in the former case the environment of the caller must be that of the (known) called function, possibly with additional values added on. This is a consequence of lexical scoping. As a result, the function call can be compiled as a GOTO preceded by an environment adjustment which consists merely of lopping off some leading portion of the current one (intuitively, one simply "pops the unnecessary crud off the stack"). LABELS-closed functions also can be treated in this way, if one closes all the functions in the same way (but this is not always desirable). If one does, then it is easy to see the effect of expanding a PROG into a giant LABELS as outlined in [Steele 1976A] and elsewhere: normally, a GOTO to a tag at the same level of PROG will involve no adjustment of environment, and so compile into a simple GOTO instruction, whereas a GOTO to a tag at an outer level of PROG probably will involve adjusting the environment from that of the inner PROG to that of the outer. All of this falls out of the proper imperative treatment of function calls.

Optimization

The previous section, in order to elucidate the issue of tail-recursion, showed optimizations as if they were performed at

the target-language level. In fact, the necessary optimizations are performed at the source level. Once one realizes that there is a direct correspondence between function calls and "goto" statements, there is no problem with representing arbitrarily complex control structures at the source level.

RABBIT uses only about a half dozen types of source-to-source transformations. Some of these are fairly obvious:

```
((LAMBDA () body))  =>  body
```

Others are well-known transformations, such as substituting arguments for formal parameters in LAMBDA-expressions, or eliminating unreferenced parameters, or dead code elimination. The one interesting and not-widely-known transformation involves nested IF expressions. The basic idea is:

```
(IF (IF a b c) d e)  =>  (IF a (IF b d e) (IF c d e))
```

One problem with this is that the code for d and e is duplicated. This can be avoided by the use of LAMBDA-expressions:

```
((LAMBDA (Q1 Q2)
         (IF a
             (IF b (Q1) (Q2))
             (IF c (Q1) (Q2))))
  (LAMBDA () d)
  (LAMBDA () e))
```

While this code may appear unnecessarily complex, the calls to the functions Q1 and Q2 will, as shown above, be compiled as simple GOTO's. As an example, consider the expression:

```
(IF (AND PRED1 PRED2) (PRINT 'WIN) (ERROR 'LOSE))
```

Expansion of the AND macro will result in an IF expression. Applying the transformation above and then simplifying produces:

```
((LAMBDA (Q2)
         (IF PRED1
             (IF PRED2 (PRINT 'WIN) (Q2))
             (Q2)))
  (LAMBDA () (ERROR 'LOSE)))
```

Recalling that (Q2) is, in effect, a GOTO branching to the common piece of code, and that by virtue of later analysis no actual closure will be created for either LAMBDA-expression, this result is quite reasonable. It does not evaluate the second predicate if the first results in NIL (false). The optimization on nested IF expressions has resulted in a simple source-level implementation of the optimization known as "anchor pointing" or "evaluation for control." (In most compilers this is handled at the code generation level by routines that pass around tags to jump to on success or failure of a predicate.)

This is a phenomenon we have noticed several times in RABBIT: because the many control constructs have been expressed in terms of only a few, only a few general and powerful optimizations techniques are needed, which tend to combine to produce more traditional techniques as special cases. (Another example of this phenomenon is that loop unrolling falls out as a special case of parameter substitution in LABELS statements.)

Environment and Closure Analysis

Just before the code generation stage, RABBIT determines for each LAMBDA-expression whether a closure will be needed for it at run time. The idea is that in many situations (particularly those generated as expansions of macros) one can determine at compile time precisely which function will be invoked by a combination, and perhaps also what its expected environment will be. There are three possibilities:

(1) If the function denoted by the LAMBDA-expression is bound to some variable, and that variable is referenced other than in

function position, then the closure is being treated as data, and must be a full (standard CBETA format) closure. If the function itself occurs in non-function position other than in a LAMBDA-combination, it must be fully closed.
(2) If the closure is bound to some variable, and that variable is referenced only in function position, but some of these references occur within other partially or fully closed functions, then this function must be partially closed. By this we mean that the environment for the closure must be "consed up," but no pointer to the code need be added on as for a full closure. This function will always be called from places that know the name of the function and so can just perform a GO to the code, but those such places which are within closures must have a complete copy of the necessary environment.
(3) In other cases (functions bound to variables referenced only in function position and never within a closed function, or functions occurring in function position of LAMBDA-combinations), the function need not be closed. This is because the environment can always be fully recovered from the environment at the point of call. The typical case of this is LABELS functions, which are seldom passed as parameters and which are all defined in the same environment. Because no run-time closures are needed for them, there is no run-time cost associated with entering or leaving a LABELS expression; because they are defined in the same environment, there is usually no environment adjustment needed for one to call another. Using LABELS functions to express GOTO or iteration entails no unexpected overhead.

In order to determine the closure information, it is necessary to determine, for each node, the set of variables referred to from within closed functions at or below that node. Thus this process and the process of determining which functions to close are highly interdependent, and so is accomplished in a single pass.

Conclusions

Lexical scoping, tail-recursion, the conceptual treatment of functions (as opposed to representations thereof) as data objects, and the ability to notate "anonymous" functions make SCHEME an excellent language in which to express program transformations and optimizations. Imperative constructs are easily modelled by applicative definitions. Anonymous functions make it easy to avoid needless duplication of code and conflict of variable names. A language with these properties is useful not only at the preliminary optimization level, but for expressing the results of decisions about order of evaluation and storage of temporary quantities. These properties make SCHEME as good a candidate as any for an UNCOL. The proper treatment of functions and function calls leads to generation of excellent imperative low-level code.

We have emphasized the ability to treat functions as data objects. We should point out that one might want to have a very simple run-time environment which did not support complex environment structures, or even stacks. Such an end environment does not preclude the use of the techniques described here. Many optimizations result in the elimination of LAMBDA-expressions; post CPS-conversion analysis eliminates the need to close many of the remaining LAMBDA-expressions. One could use the macros and internal representations of RABBIT to describe intermediate code transformations, and require that the final code not actually create any closures. As a concrete example, imagine writing an operating system in SCHEME, with machine words as the data domain (and functions excluded from the run-time data domain). We could still meaningfully write, for example:

```
(IF (OR (STOPPED (PROCESS I))
        (AWAITING-INPUT (PROCESS I)))
    (SCHEDULE-LOOP (+ I 1))
    (SCHEDULE-PROCESS I))
```

While the intermediate expansion of this code would conceptually involve the use of functions as data objects, optimizations would reduce the final code to a form which did not require closures.

References

Frances E. Allen and John Cocke, "A Catalogue of Optimizing Transformations," in Randall Rustin (ed.), *Design and Optimization of Compilers*, Proc. Courant Comp. Sci. Symp. 5, Prentice-Hall, 1972.

Daniel G. Bobrow and Ben Wegbreit, "A Model and Stack Implementation of Multiple Environments," *CACM* Vol. 16, No. 10, 1973.

Alonzo Church, *The Calculi of Lambda Conversion*, Annals of Mathematics Studies Number 6, Princeton University Press, 1941, Reprinted by Klaus Reprint Corp., 1965.

Samuel S. Coleman, *JANUS: A Universal Intermediate Language*, PhD thesis, University of Colorado, 1974.

Edsger W. Dijkstra, *A Discipline of Programming*, Prentice-Hall, 1976.

Jon Doyle, Johan de Kleer, Gerald Jay Sussman, and Guy L. Steele Jr., "AMORD: Explicit Control of Reasoning," Proc. AI and Programming Languages Conf., SIGPLAN Notices 12, 8, SIGART Newsletter 64, 1977.

Charles M. Geschke, *Global Program Optimizations*, PhD thesis, Carnegie-Mellon University, 1972.

David Gries, *Compiler Construction for Digital Computers*, John Wiley and Sons, 1971.

Peter J. Landin, "A Correspondence between ALGOL 60 and Church's Lambda-Notation," *CACM* Vol. 8, No. 2-3, 1965.

John McCarthy *et al.*, *LISP 1.5 Programmer's Manual*, The MIT Press, 1962.

David A. Moon, *MACLISP Reference Manual, Revision 0*, MIT Laboratory for Computer Science, 1974.

Joel Moses, *The Function of FUNCTION in LISP*, MIT AI Laboratory Memo 9, 1970.

John C. Reynolds, "Definitional Interpreters for Higher Order Programming Languages," *ACM Conference Proceedings*, 1972.

Jean E. Sammet, *Programming Languages: History and Fundamentals*, Prentice-Hall, 1969.

T. A. Standish *et al.*, *The Irvine Program Transformation Catalogue*, University of California, 1976.

Guy L. Steele Jr., "Debunking the 'Expensive Procedure Call' Myth," submitted to the 77 ACM National Conference, 1977A.

Guy Lewis Steele Jr., *Compiler Optimization Based on Viewing LAMBDA as RENAME Plus GOTO*, SM Thesis, MIT, 1977B.

Guy Lewis Steele Jr. and Gerald Jay Sussman, *LAMBDA: The Ultimate Imperative*, MIT AI Laboratory Memo 353, 1976A.

Guy Lewis Steele Jr., *LAMBDA: The Ultimate Declarative*, MIT AI Laboratory Memo 379, 1976B.

Joseph Stoy, *The Scott-Strachey Approach to the Mathematical Semantics of Programming Languages*, MIT Laboratory for Computer Science, 1974.

Warren Teitelman, *InterLISP Reference Manual* Revised edition, Xerox Palo Alto Research Center, 1975.

Mitchell Wand, and Daniel P. Friedman, *Compiling Lambda Expressions Using Continuations*, Technical Report 55, Indiana University, 1976.

William A. Wulf, et al., *The Design of an Optimizing Compiler*, American Elsevier, 1975.