Design of An Optimizing, Dynamically Retargetable Compiler for Common Lisp

Rodney A. Brooks

Massachusetts Institute of Technology
Lucid, Inc.

David B. Posner, James L. McDonald, Jon L. White
Eric Benson, Richard P. Gabriel

Lucid, Inc.

Abstract

We outline the components of a retargetable cross-compiler for the Common Lisp language. A description is given of a method for modeling the various hardware features in the compiler's database, and a breakdown is shown of the compiler itself into various machine-independent and machine-dependent modules. A novel feature of this development is the dynamic nature of the retargeting: Databases for multiple hardware architectures are a standard part of the compiler, and the internal interfaces used by the compiler are such that the machine-dependent modules may be instantly switched from one to another. Examples of generated code in several environments will be given to demonstrate the high quality of the output available, even under this very modular approach.

1. Introduction

This paper describes the techniques used to develop the Lucid Portable Common Lisp compiler. The major focus of this paper is on the problems and techniques relating to the goals of producing a compiler which is retargetable and, at the same time, capable of producing highly tailored code for each target architecture. The compiler that resulted from this effort is not only retargetable, but dynamically retargetable in the sense that multiple target descriptions may reside in the same runtime environment, and the desired target is passed as an optional argument to the compiler.

The major techniques used are (1) the characterization of virtual machine classes as classes of code transformations and (2) an object-oriented representation of these machine classes which exploits inheritance to maximize the sharing of code and data across targets. The refinement of these techniques has led us to use 'derived constants' and other forms of parametric data.

2. Target Model

This section describes the attributes of the machine architecture and the Lisp implementation design that are modeled by the compiler. We develop a framework for modeling machine architecture that facilitates the particular code-generation and optimization techniques we use. Some modeled attributes of the target machine architecture are:

1. whether or not there is a hardware stack pointer and a push instruction
2. whether data movement can be memory/memory, or is limited to memory/register (as in RISC machines)
3. how registers are grouped into classes (such as 'address' and 'data')
4. how the component slots of a stack frame are accessed
5. how other locations are accessed, including local variables, special value cells of dynamic variables, and other stack frame slots
6. what side effects the instructions and pseudo-instructions have.

In addition to machine-specific items, there is another set of features which are implementation dependent; given a particular hardware architecture, it is possible to devise several Lisp implementations for that hardware. While these features may be motivated by particular concerns for the hardware capabilities, they are not direct consequences of it, but rather are consequences of our choices when implementing the Lisp. Some of the more important of these attributes are

1. The primary tagging scheme—where do the tags go?
2. The stack frame layout—which slots hold machine-dependent overhead, which hold local variables, what information is passed during function call?
3. Which primitive operations will be open-coded into native machine instructions, and which will be coded as machine language subroutines? We use the term 'fastcall' for this level of hand-coded subroutine. Because these subroutines do not require building a Lisp stack frame, they use the most efficient subroutine linkage on the machine.

3. Organization of the compiler

There are four phases in the compiler: (1) alphatization, (2) analysis, (3) annotatation, and (4) code generation. The structure of the compiler is similar to that of the S1 Lisp compiler [Brooks 1982a]. Each phase may exploit machine-dependent information.

In brief, the analysis phase of the compiler performs source-to-source transformations. These transformations actually operate on an internal, graph-like data structure, but the transformations correspond to transformations the user could have performed himself.

The phases proceed sequentially except that the analysis phase can reinvoke alphatization after it has done a source-to-source transform. Usually the reinvocation concerns only that part of the original code that has been changed.

3.1 *Alphatization*

The alphatization phase translates the input code into a subset of Common Lisp. The result, however, is very much like a valid Common Lisp program, perhaps referring to some functions not specified in Common Lisp, but defined within Lucid Common Lisp. For example, the function TRIANGLE in Figure 1 is alphatized to the code shown in Figure 2.

```
(defun triangle (n)
 (labels ((tri (c n)
           (declare (type fixnum n c))
           (if (zerop n)
               c
               (tri (the fixnum (+ n c))
                    (the fixnum (- n 1))))))
   (tri 0 n)))
```

Figure 1: The source code for a procedure to compute triangular numbers. Declarations have been added to force fixed precision arithmetic. Notice that it is not sufficient to declare N and C to be fixed precision, we must also declare that their sum does not overflow.

```
(LAMBDA (N)
 (&LET ((G1851 'NIL))
  (&LET
   ((G1853
     #'(INTERNAL-LAMBDA G1852 (C N-1)
        (LOCALLY
         (DECLARE (TYPE FIXNUM N-1) (TYPE FIXNUM C))
         (BLOCK G1854
          (TAGBODY
           (IF (FIXNUMP N-1)
               (IF (FIXNUM-COMPARE-ZERO N-1 'EQ)
                   (GO G1855)
                   (GO G1856))
               (IF (GENERIC-ZEROP N-1) (GO G1855) 'NIL))
           G1856
           (RETURN-FROM G1854
            (&FUNCALL G1851 (THE FIXNUM (+ N-1 C))
                            (THE FIXNUM (- N-1 '1))))
           G1855
           (RETURN-FROM G1854 C)))))))
   (PROGN (SETQ G1851 G1853)
          (&FUNCALL G1853 '0 N)))))
```

Figure 2: The code alphatized for the Motorola MC68000 as a target machine. Variable names are unique. The LABELS construct has turned into more primitive operations with LAMBDA expressions and FUNCALL's. One generic arithmetic predicate, ZEROP, has been expanded into an inline specialization for fixed precision arithmetic.

The function &FUNCALL is like Common Lisp FUNCALL, except that it assumes its first argument is a valid procedure object and does neither coercion nor error checking. &LET is like LET, except that no declarations are allowed in the body, which must be a single s-expression.

Specific operations carried out during alphatization include:

- Macro expansion. This involves both user macros and internal compiler macros. Some compiler macros implement Common Lisp special forms in terms of others. Other compiler macros translate Common Lisp functions to instances of more general Lucid primitive operations ('primops'): for instance, CAR, CDR, SYMBOL-NAME, SYMBOL-VALUE, structure references, and several other operations all compiler-macro-expand into a pointer dereference operation, %POINTER-REF-CONSTANT, with some constant offset.

- Uniquification of variable names.[1] This entails maintaining a lexical environment while the source code tree is walked. Later we will see that alphatization can be re-invoked from the analysis phase, so alpha phase lexical environments must be compatible with analysis environments. Unique variable names make many later substitutions simpler because there is no concern that variables have been shadowed at a point where a substitution is being made.

- Manipulation of declarations. Declarations are not of interest during alphatization, but great care must be taken as code is transformed to ensure that the transformed declarations maintain the originally intended scope. The exceptions are that special declarations can be processed and discarded during this phase, and that NOTINLINE declarations must be processed and still maintained in case analysis reinvokes alphatization. For instance, when declared NOTINLINE, CAR might be stopped from being expanded into an unsafe operation, and later a call to the error checking version will be emitted.

- Constant folding. Declared constants are folded into the code; procedure calls with literal arguments are evaluated at compile time, and the results are folded into the code, provided that the procedure is known neither to have side effects nor to be affectable by side effects. Constants are not folded when EQL semantics might be violated. For example, if the constant L is a list, folding might make (EQL L L) be false, whereas if L is a number, then the folding will preserve the EQL relation. If the predicate clause of an IF special form can be evaluated then dead code can also be eliminated. Such code most frequently arises as a result of macro-expansion.

- Handling of special variable references. In shallow binding Lisps such references are turned into calls to the Lisp primitive SYMBOL-VALUE. This turns into an instance of a general structure reference.

- Quotation of self evaluating forms. This makes later analysis simpler.

- Reduction to simplest syntatic cases. For example, the implicit PROGN's in LET and UNWIND-PROTECT are made explicit.

[1] Contrast this with 'liquification,' in which lexical variables are made fluid.

- Number of arguments error checking. The compiler has a data base of the valid numbers of arguments for all Common Lisp functions and extends this database whenever it compiles a user function. During alphatization the user is warned when a function is called with the wrong number of arguments or when a function is redefined with a different number of arguments.

- Handling of local functions. Local functions are turned into explicit first-class data objects. Later, such objects may get optimized into inline code.

- Application of target-specific optimizers. Each primop for a machine may optionally have a source-to-source optimizer associated with it. For instance, on some machines, extracting a character from a string at some fixed index can be turned into a move byte instruction with a simple operand. However, the range of indices over which this instruction and addressing mode will work is usually limited in a machine-dependent way. An optimizer attached to the SCHAR primop converts it to a more specific primop when appropriate.

3.2 *Analysis*

The analysis phase walks over the alphatized code, which is now made up of, essentially, a subset of full Common Lisp. As it descends the code, tree it builds a parallel analysis tree. Each Lisp combination and terminal (variable or literal) has an ANODE associated with it.

A representation of the lexical environment is built as the code tree is descended. Variable, tag, and block references, along with type declarations, are recorded in these structures.

An ANODE contains a number of slots. They include:

- an internal combination-specific subnode. There is one such subnode class for each special form, one for procedure calls, one for primops, and one for variable references. These subnodes contain pointers to the appropriate subtrees.

- a list of ANODE's that can provide the value this ANODE returns. Sometimes the only member of this list is the ANODE itself—for instance, for an ANODE corresponding to a variable reference. For an ANODE associated with an IF special form, however, there will be at least two entries and often more: There will be all the return ANODE's from the 'then' clause and all those from the 'else' clause.

- the source code associated with this node. All the source slots of descendant nodes share the structure of this code appropriately so that source level substitutions propagate throughout the tree appropriately.

- the number of items in the current stack frame when the code for this ANODE is entered.

- the side effects of this ANODE. This has two components: (a) the Lisp-level side effects, and (b) the number of each class of registers (e.g., marked, unmarked, or, on the Motorola MC68O2O, marked address registers, marked data registers, etc.) which must be used by this code even if no variables reside in registers. There are three possibilities for Lisp-level side effects:

  NIL: This ANODE is neither affected by nor affects any other computation. This makes it a prime candidate for being computed out of order, if that provides some optimization.

VAL: This ANODE can be affected by other code with explicit side effects, or it CONSes to produce its result.

T: This ANODE has explicit side effects. The worst case must always be assumed so a call to an unknown procedure always produces such a side effects description.

- the desired location for the result of this ANODE.
- an operand describing where the result of this ANODE will be.
- the most restrictive type known for the result of this ANODE. This is deduced from a combination of type declarations concerning the ANODE itself and from taking the union of types of all the value-producing ANODEs for this one.

When a tree consisting of only primops, variables, and literals is analyzed, the primops can pass results to their parents directly in registers. The side effects slot is compared to the known number of registers of each class, and, when there are not enough, a source-to-source transform is done to introduce, via LET, a temporary variable to hold the result. Such a new variable will inevitably end up being allocated in a stack location, solving the register shortage. A little care must be taken so that the optimizations listed below do not inadvertantly undo such transformations leading to infinite reanalysis loops.

As the analysis tree is built and analysis is re-ascending, many optimizations can be done. These correspond to the Beta Reductions of Steele's RABBIT compiler [Steele 78]. Some optimizations can be achieved by patching the analysis tree (and maintaining a compatible source), while others require realphatization and reanalysis of the source code associated with the current subtree.

Some (but not all) of the machine independent optimizations currently performed include:

- Trees of PROGN's are flattened into a single PROGN.
- Non-final clauses of a PROGN that have side effect class NIL or VAL are eliminated.
- Unreferenced BLOCK's are eliminated.
- Unreferenced tags in TAGBODYs' are eliminated.
- TAGBODY's with no tags are replaced by PROGN's with a final clause of 'NIL.
- Variables bound to constants and never SETQed are replaced by the constants.
- Variables bound to variables are eliminated if there are no SETQ's that would make such substitutions invalid.
- Variables bound but never read are eliminated. The forms to which they are bound are also eliminated if they have side effects of class NIL or VAL.
- Variables that are referenced only once in code that is executed only once relative to the binding of the variable are eliminated if the form to which they are bound can be moved past intervening forms without any side effects interactions.

- Lambda expressions that only get used once and then only in a FUNCALL that was not generated from a call to a LABELS or FLET function that was declared NOTINLINE get substituted inline.
- Lambda expressions that only get used on downward calls and that make lexical references outside their own scope have an argument added which is a stack display register.
- Self tail recursive calls are turned into jumps.
- Non-self tail recursive calls are reorganized so that existing stack locations are SETQed to the arguments being passed in such a way that the existing stack frame can be re-used by the called procedure. The case of a tail recursive FUNCALL requires special treatment.

The following code shows the results of these and other optimizations to the code in Figure 1:

```
(LAMBDA (N)
 (&LET ((C '0) (N-1 N))
  (LOCALLY (DECLARE (TYPE FIXNUM N-1)
                    (TYPE FIXNUM C))
  (BLOCK G1854
   (BLOCK G1911
    (TAGBODY
     (GO G1908)
     G1915 (PROGN
            (SETQ C (THE FIXNUM (+& C N-1)))
            (SETQ N-1 (THE FIXNUM (-& N-1 '1)))
            (GO G1908))
     G1908 (IF (FIXNUM-COMPARE-ZERO N-1 'EQ)
               (GO G1914)
               (GO G1915))
     G1914 (RETURN-FROM G1911
            (RETURN-FROM G1854 C))))))))
```

Figure 3: The code after optimizations during analysis. Type propagation has eliminated the GENERIC-ZEROP. Self tail recursion of TRI has turned into a GO (to label G1908). The resulting loop has then been optimized to have the test at the end of the loop rather than at the beginning. TRI has been substituted inline.

3.3 *Annotation*

The annotation phase deals with storage allocation for variables. It actually consists of two independent walks over the analysis tree.

3.3.1 *Phase 1*

At each node in the tree the possibility of individually assigning each variable to a register for the life of the code represented by the node is analyzed. In particular, the impact of such an allocation in the number of side-effects registers is analyzed—many times the quantity would already have to be in a register at some point in the subtree, so it is not simply a matter of adding a new register to the already required registers. Some elementary flow analysis is also done.

3.4 *Phase 2*

In the second tree walk each introduced variable is allocated either to a stack location or to a register. Variables allocated to a register when they are introduced remain in that register for their lifetimes.

Additionally, as each ANODE is reached, a check is done for any variables that may already be assigned a stack location and that might profitably be cached into an available register for the duration of the code of the ANODE. The variable-reference records built during analysis and during the elementary flow analysis done are used to determine whether the register needs to be decached to the stack on exit from the ANODE, and exactly what forms the exits may take (e.g., GO's to tags outside the scope of the current ANODE).

3.5 *Code generation*

The code generator walks over the analysis tree, allocating specific registers to variables that need them.

For a special form ANODE or a general procedure call, the generator outputs calls to LAP macros. The arguments can vary in a machine-dependent way.

Primops are expanded using the code and result templates associated—machine-dependently—with the primop. As instructions are generated, a final peephole phase eliminates redundant jumps and other simple inefficiencies. Figure 4 shows the results of all four phases of the compiler on TRIANGLE, introduced in Figure 1:

```
  (SIMPLE-ENTRY 1)     ; check for exactly 1 args
START
  (MOVE (QUOTE 0) D0)  ; install let var C
                       ; ref var N
  (MOVE (FRAME 3) D1)  ; install let var N-1
  (JMP G1943)          ; go to tag: G1908
G1942
                       ; primop +&
                       ; setq C
  (ADD D1 D0)          ; ref var N-1
                       ; primop -&
  (SUB (QUOTE 1) D1)   ; setq N-1
G1943
  (CMP (QUOTE 0) D1)   ; primop FIXNUM-COMPARE-ZERO
  ((BRANCH NE) G1942)  ; branch on falseness
  (MOVE D0 (FRAME -1)) ; ref var C
  (SIMPLE-EXIT)
```

Figure 4: The generated code for the MC68000. Notice that the nested RETURN-FROM's have disappeared. They were handled through coercion of the operand specifying where C could be found to the destination of BLOCK G1854, which is returned from the procedure. Additionally note that although ADD instructions are used, the representation of fixed precision integers is not machine numbers but rather the machine number 4 times the size of the Lisp number. This trick lets 30 bit integers be immediate quantities within a pointer.

3.6 *A Second Example*

Figures 5 through 9 illustrate the compilation of the same source code for two different targets: the Motorola MC68020 and the PRIME 50 Series

The source code is:

```
(defun bazola (a b)
  (let ((n 7)
        (c a))
    (cond ((compiled-function-p c) (cadr b))
          (t (setf (cdr a) (svref b n))
             '()))))
```

Figure 5: The source code to be compiled for both the MC68020 and the PRIME 50 Series.

The results diverge significantly during alphatization. For the MC68020, the tagging scheme is such that COMPILED-FUNCTION-P needs only work with the supplied pointer. On the PRIME it is necessary to follow the pointer and examine the object itself. There are other minor differences in the calls to %POINTER-REF-CONSTANT due to variations in the tagging scheme used.

```
(LAMBDA (A B)
  (BLOCK G28652
    (PROGN
     (IF (%COMPARE-FIELD-CONSTANT A '0 '3 '7 'EQ)
         (IF (LOGTEST&-PRIMOP '64 (%POINTER-REF-CONSTANT A '7 '3) 'EQ)
             (RETURN-FROM G28652
               (%POINTER-REF-CONSTANT (%POINTER-REF-CONSTANT B '1 '0) '1 '1))
             'NIL)
         'NIL)
     (SET-%POINTER-REF-CONSTANT A '1 '0 (%POINTER-REF-CONSTANT B '3 '8))
     'NIL)))
```

Figure 6: BAZOLA alphatized and analyzed for the MC68020.

```
(LAMBDA (A B)
  (IF (BLOCK G28659
        (PROGN
         (IF (%COMPARE-FIELD-CONSTANT A '0 '2 '2 'EQ)
             (IF (%COMPARE-FIELD-CONSTANT (%POINTER-REF-CONSTANT A '2 '0)
                                          '0 '16 '2148 'EQ)
                 (RETURN-FROM G28659
                  (LOGTEST&-PRIMOP '64 (%POINTER-REF-CONSTANT A '2 '3) 'EQ))
                 'NIL)
             'NIL)
         'NIL))
      (%POINTER-REF-CONSTANT (%POINTER-REF-CONSTANT B '1 '0) '1 '1)
      (PROGN
       (SET-%POINTER-REF-CONSTANT A '1 '0 (%POINTER-REF-CONSTANT B '2 '8))
       'NIL)))
```

Figure 7: BAZOLA alphatized and analyzed for the PRIME.

In both cases, analysis eliminated variables N and C.

```
  (SIMPLE-ENTRY 2)      ; check for exactly 2 args
START
  (MOVE (FRAME 3) A0)   ; cache A
  (MOVE A0 D0)          ; ref var A
  ((AND B) 7 D0)
  ((CMP B) 7 D0)        ; primop %COMPARE-FIELD-CONSTANT
  ; branch on falseness
  ((BRANCH NE) G28655)  ; ref var A
  (MOVE (POINTER-REF-CONSTANT A0 7 3) D0) ; primop %POINTER-REF-CONSTANT
  (MOVE (QUOTE 64) D1)
  (AND D0 D1)   ; primop LOGTEST&-PRIMOP
  ((BRANCH NE) G28656)  ; branch on falseness
  (MOVE (FRAME 4) A1)   ; ref var B
  (MOVE (POINTER-REF-CONSTANT A1 1 0) A1) ; primop %POINTER-REF-CONSTANT
  ; primop %POINTER-REF-CONSTANT
  (MOVE (POINTER-REF-CONSTANT A1 1 1) (FRAME -1))
  (SIMPLE-EXIT)
G28656
G28655
  ; ref var B
  ; primop %POINTER-REF-CONSTANT
  (MOVE (FRAME 4) A1)   ; ref var A
  ; primop SET-%POINTER-REF-CONSTANT
  (MOVE (POINTER-REF-CONSTANT A1 3 8) (POINTER-REF-CONSTANT A0 1 0))
  (MOVE (QUOTE NIL) (FRAME -1))
  (SIMPLE-EXIT)
```

Figure 8: The Motorola BAZOLA code.

Finally, in code generation we see a reflection of the differing numbers of available marked address registers on the MC68020 versus the PRIME. On the MC68020 there are only two such non-dedicated registers. One is needed to hold intermediate values (e.g., between the CDR and the CAR of the CADR). Thus only argument A is cached into a register, while B remains on the stack. On the PRIME, however, there are more available registers and both variables get cached into registers. Note the difference in the legal combinations of arguments to a MOVE instruction causing the SETF (i.e., the SET-%POINTER-REF-CONSTANT) to take two instructions in the PRIME, but only one on the MC68020. Note that the implementations for the different machines handle return values differently. In the MC68020 implementation, values are returned on the stack, while on the PRIME they are returned in a register.

```
  (SIMPLE-ENTRY 2 4)  ; check for exactly 2 args
START
  (MOVE (FRAME 4) R1) ; cache B
  (MOVE (FRAME 3) R2) ; cache A
  (MOVE R2 R6)        ; ref var A
  (AND 3 R6)
  (CMP 2 R6)          ; primop %COMPARE-FIELD-CONSTANT
  ; branch on falseness
  ((BRANCH NE) G28664); ref var A
  ; primop %POINTER-REF-CONSTANT
  (MOVE (POINTER-REF-CONSTANT R2 2 0) R6)
  (ICHR R6)
  ((CMP H) 2148 R6)  ; primop %COMPARE-FIELD-CONSTANT
  ; branch on falseness
  ; ref var A
  ((BRANCH NE) G28665) ; primop %POINTER-REF-CONSTANT
  (MOVE (QUOTE 64) R3)
  (AND (POINTER-REF-CONSTANT R2 2 3) R3)
  (CMP (QUOTE 0) R3)
  (COERCE-FLAG-TO-POINTER EQ R4) ; primop LOGTEST&-PRIMOP
  (JMP G28663)       ; exit block: G28659
G28665
G28664
  (MOVE (QUOTE NIL) R4)
G28663
  (COMPARE-TO-NIL R4)
  ; branch on falseness
  ((BRANCH EQ) G28661)   ; ref var B
  ; primop %POINTER-REF-CONSTANT
  (MOVE (POINTER-REF-CONSTANT R1 1 0) R3)
  ; primop %POINTER-REF-CONSTANT
  (MOVE (POINTER-REF-CONSTANT R3 1 1) R1)
  (SIMPLE-EXIT)
G28661
  ; primop %POINTER-REF-CONSTANT
  (MOVE (POINTER-REF-CONSTANT R1 2 8) R3) ; ref var A
  ; primop SET-%POINTER-REF-CONSTANT
  (MOVE R3 (POINTER-REF-CONSTANT R2 1 0))
  (MOVE (QUOTE NIL) R1)
  (SIMPLE-EXIT)
```

Figure 9: PRIME code generated from BAZOLA.

3.7 *LAP—the Lisp Assembler Program*

The LAP assembler takes lap code as shown in Figures 8 or 9 and produces a procedure object including referenced constant forms plus a vector of bits representing machine code for the target machine. Each form in the lap source is either a label, a lap instruction, or a lap macro. The top level of LAP is machine-independent and makes essentially one pass over the source lap code, maintaining data pertinent to the optimized resolution of branches and jumps. In an attempt to minimize the work needed to create the assembler for a new machine, most of the algorithms for LAP have been made essentially machine independent, including those for expanding lap macros, resolving labels, and optimizing branches. As much as possible, the machine-dependent information exists in tabular form. The primary exception to this scheme involves machine-dependent methods used by the branch resolution algorithm. There are about 80 pages of source code for machine-independent algorithms, the same for machine-dependent declarations and code, and about 120 pages of machine-dependent opcode tables and lap macros. It takes from two weeks to a month to implement LAP for a new processor, including all the macros used to implement the compiler's virtual machine.

3.7.1 *Macro Expansion*

LAP macros for both instructions and operands are strongly analogous to normal Common Lisp macros, except that those for instructions expand into a sequence which is spliced into place. Figure 10 shows how SIMPLE-ENTRY is defined for the MC68000 machine. Note that FASTCALL is also a lap-macro which is expanded in turn. Figure 10 also shows how FRAME is defined to expand into a more explicit operand.

```
(deflap-macro simple-entry (n)
  `((((cmp w) ,n countreg)
    ((branch eq) start)
    (fastcall ,(target sq-incorrect-no-args))))
```

```
(deflap-operand frame (offset)
  `(ref fp ,(* (target frame-pointer-size) offset)))
```

Figure 10: Definitions of a LAP instruction macro and a LAP operand macro.

3.7.2 *Operand Decoding*

After all macros are expanded, the operands are decoded using a combination of machine-independent and machine-specific routines. For each operand, its lap operand type is determined, and component features are extracted and stored in a lap structure. For instance, for the MC68000, a stack frame reference such as '(REF FP −24)' would be assigned type LAP-TYPE-AREG-INDIRECT-WITH-DISP, with address register 6 and displacement −24. The assigned lap operand type may be the union of more primitive types.

3.7.3 *Table-Driven Code Generation*

Once the operands have been decoded, a machine-specific procedure associated with the opcode is called either to generate an actual bit pattern, or, when labels are referenced, to call a routine which resolves such references. The machine-specific opcode procedures are created via special defining macros that admit terse descriptions of how instructions expand. These macros create code generators that select among alternate machine instructions based upon the types of the operands. When targeting the MC68000, for example, (MOVE 0 A0) becomes (SUBA A0 A0) rather than (MOVEA 0 A0), since SUBA in this case has the same effects (including those on the condition code), but runs faster.

3.7.4 *Reference Resolution and Branch Optimization*

The difficult part of LAP involves resolving references to labels, since instructions may vary in length according to the location of the label they reference, and this variance may affect the location of the label. Thus, in general, some sort of relaxation technique is needed to arrive at optimal code. Leverett [Leverett 1980] and others have presented algorithms that optimize programs for code density by converting long jumps into short jumps to long jumps. Since we optimize for time rather than space in such cases, these algorithms were not found appropriate.

Since machines vary greatly in their referencing modes, it was not feasible to write a totally machine-independent optimizing reference resolver. It was possible, however, to factor most of the algorithm into high-level machine-independent routines that invoke machine-specific methods for answers to particular questions. In particular, these methods are given a specific reference in a specific context and either resolve it immediately or return the minimum and maximum sizes the reference could require. There is also a 'catch-all' machine-specific polling routine that is invoked whenever a machine specific condition is met. On machines with base addressing, the poll function handles the transition from one region of addressibility to the next. On the MC68000, this routine could be used to notice that a reference to an as-yet-unseen label is now followed by at least 126 bytes of code; hence the reference must be of the long form and can be profitably reanalyzed.

3.8 *FASL—Conversion of Compiled Programs to Binary File Format*

The compiler/assembler produce output files in a specially designed format. This file format can store arbitrary Lisp data. The output of the compiler/assembler is a graph—circularities are permitted. The file format is expressive enough to capture the graph, so that a graph isomorphic to the original is created when the file is loaded into a different invocation of Lisp. The format is the same for all implementations of Lucid Common Lisp, so that the programs which read and write these files are almost completely machine-independent. However, code objects can only be expected to run on the machine for which they were compiled.

4. Dynamic Retargeting

The compiler is dynamically retargetable in the sense that descriptions of multiple target machines can coexist in the compiler and the choice of a particular target is passed as an optional argument to COMPILE-FILE, as in

```
(compile-file "foo" :target 'prime)
(compile-file "foo" :target 'sun3)
(compile-file "foo" :target 'apollo)
```

We refer to the act of altering the machine for which the compiler compiles as 'changing the target' of the compiler.

Because of the size and complexity of the compiler we have attempted to maximize the amount of code and data sharing among targets without inhibiting the ability to utilize arbitrarily specific target-dependent information at all stages of the compilation process.

4.1 *Parameterization of the compiler.*

We briefly summarize the various kinds of target-specific information required by the compiler at the different phases described earlier.

4.1.1 *Alphatization*

Target-dependent information employed during alphatization includes constants, compiler macros, and code optimizers. Target-dependent constants required during this phase include both external constants such as MOST-POSITIVE-FIXNUM and internal constants such as constants for the primary and secondary tag values, and header and data offsets for various types of structures. The treatments of COMPILED-FUNCTION-P during alphatization in BAZOLA illustrate the use of target-dependent compiler macros to implement differing tag schemes.

Differing reduction strategies may also be implemented in this way. For example, one target might use a compiler macro to reduce one control-structure to another (e.g. CASE to COND) where another might provide a more direct implementation. Code optimizers may differ, for example, because of differences in hardware capabilities. For example, differences in the availability of indexed addressing modes and the corresponding range capabilities may be reflected in different optimizers for structure accessing.

4.1.2 *Analysis*

The primary use of target-dependent information during the analysis phase is involved in register analysis. For each target machine the register analyzer requires a specification of the register classes defined for that machine, together with the registers available in each class. For example, for the MC68000, the register classes include 'address-registers' and 'data-registers' and among the address registers are registers A0, A1, etc.[2] Each primop definition includes a specification of the register requirements of the primop. In particular, this information includes what register classes (including memory) are possible for each argument, what additional registers are required and to what classes they may belong, what registers may be modified by the primop, and where the value or values of the primop are returned. In addition to this information, the register analyzer needs to know whether, for the given target, values are returned in registers, and if so, which ones. Type analysis may also involve target-dependent information, as when determining whether the fixnum version of an operator may be applied to an expression involving an integer literal.

4.1.3 *Annotation*

The annotation phase requires essentially the same target-dependent information about registers and register classes as is required during analysis. In addition, target-dependent constants defining the stack frame format are required in order to assign specific frame locations for variables.

4.1.4 *Code Generation*

Target-dependent information employed during code generation includes: the primop code and result templates; tag, flag, and structure access constants; constants defining offsets into a global data vector for fastcalled procedures and commonly accessed literals and globals; function calling and return protocols;

[2] The register allocator does not have a generic notion of 'register-class,' but, rather, each target-machine specification defines the classes appropriate for that target.

whether a stack pointer is maintained; register classes and their elements; condition codes and how to invert them (for jump optimization); target 'move' operators and their effects (for eliminating redundant move instructions).

4.1.5 *Assembly*

Target information required by LAP includes lap-macros, lap-operand-macros, operand decoders, code emitters, methods employed by the jump optimizers, and constants describing lap operand types, code formats, and address units.

4.1.6 *Target Classes*

In order to facilitate the sharing of code and data, the constants, code-transformations, methods, and other data-defining compiler targets are grouped into dependency classes. Most of the information defining a particular target is either target independent, tag-scheme dependent, or CPU dependent. Examples of target-independent information include code reductions such as reduction of COND to IF and reduction of two-dimensional array access to one-dimensional array access; optimizations such as algebraic simplification of arithmetic forms and reduction of applications of multiary operators like $+$, $*$, and NCONC, to dyadic forms; and type dispatchers for generic operators.

By the 'tag-scheme' for an implementation we mean the choice of pointer and data formats and associated tag values. Examples of data determined by the tag-scheme are MOST-POSITIVE-FIXNUM, constants defining the tag values, and the compiler macros that reduce primitive type predicates, accessors, and modifiers to bit field operations. For example, the reductions of COMPILED-FUNCTION-P, SETF, and SVREF illustrated in the function BAZOLA above are generated by macros associated with the respective tag-schemes for the two targets.

CPU-level information includes the primop expanders, register data, the availability of hardware stack support, properties of the target instruction set, and most target-specific LAP data. The additional data required to go from the analyzed computation graph to binary code for the MC68020 and PRIME examples given earlier is thus determined by CPU-level information.

In addition to the three primary dependency classes described above, other dependency classes may be defined for target attributes such as operating system and hardware characteristics beyond the CPU class.

4.2 *Object-Oriented Representation of Machine Classes*

The structuring of target descriptions into dependendency classes lends itself to an object-oriented representation of targets. The collection of data associated with a particular dependency class is bundled into a data structure called a 'compiler-machine.' There is a common compiler-machine containing all target-independent data, a compiler-machine for each tagging scheme, a compiler-machine for MC68000 class targets, a compiler-machine for MC68020 class targets, and so on. Depending on the associated dependency class, a compiler-machine may include any or all of the kinds of target-specific data described above. Thus a compiler machine may include constant values, compiler macros, primops and function descriptors, opcode-macros, operand-decoders, register-descriptors, lap-methods, and etc.

There is a natural inheritance relation defined on compiler-machines: one machine inherits from another if the class of targets associated with the first is a subset of the class of targets associated with the second.

For example, every compiler-machine inherits from the common machine, and, while it's conceivable that different tagging schemes might be used for the same CPU, that has not happened to date, and so, currently, each CPU class inherits from an associated tag-scheme class. Part of the specification of a compiler-machine is the set of compiler-machines from which it inherits.

Inheritance is implemented and exploited in the usual way. For example, the MC68020 compiler-machine inherits from the MC68000 compiler-machine except where the MC68020 extends the MC68000—for example, in bit field operations. The MC68020 compiler-machine provides compiler macros or primops which shadow those of the MC68000. As another example, the common compiler-machine provides a reduction for a case-dispatch construct, reducing it to a conditional. This reduction may be inherited during the early stages of a port and later shadowed by a more efficient CPU-level implementation which may employ a jump table. White [White 1986] gives additional examples of the use of inheritance and shadowing.

The inheritance structure that is currently implemented is hierarchical in the sense that it assumes that no conflicting values will be inherited from incomparable superclasses. Thus, any conflicts in data are resolved in favor of the more target-specific source: a CPU-specific transformation for case-dispatching will override the transform provided by the common compiler-machine. So far we have not found the need, in this context, for the more elaborate inheritance structures found in object-oriented programming systems.

4.3 *Dynamic Retargeting*

Given the explicit representation of targets as compiler-machines, the implementation of dynamic retargeting is straightforward. All accessors of target dependent information are parameterized by the 'current target,' as represented by a compiler-machine. Thus, changing targets is basically just a matter of making the compiler-machine for the new target the value of a current target global. In addition, partly for the sake of performance, internal compiler constants are bound to the values defined for the new target, so that these values can be accessed via symbol values rather than hash tables.

4.4 *Compiling the Compiler*

One of the problems when compiling the compiler has to do with the handling of target-dependent constants. The problem is that a given constant may have one value on the machine on which the compiler is actually running, a different value on the target of the compilation—that is, the machine on which the resulting compiled compiler is to be run—and still a different value for the target machine on which code produced by the compiled compiler will run. Suppose, for example, that the compiler is running on a Symbolics 3600 producing code that is to run on a SUN 3, and compiling a file of PRIME primops. One of the primops (e.g. for CONS) may mention a constant ADDRESS-UNITS-PER-CONS-CELL, which is the number of address units spanning a CONS cell.

As it happens, the values of this constant are different on all three machines—2 on the 3600, 4 on the PRIME, and 8 on the SUN 3. The problem is to determine which of the three possible interpretations of the constant is intended. We refer to this problem as the 'XYZ problem,' where X,Y, and Z stand for the three machines, X being the compiler-host, Y being the target machine (the machine on which the compiled compiler will run), and Z being the compiler-target (the target for which the compiled compiler will compile code).

On reflection, the problem turns out not to be so complicated. First, observe that the compiler host, X, is never relevant in this context. (After all, as in the example above, there is no reason that the constant need even be defined for the compiler host.) Second, in code which is not part of the compiler, the intended interpretation is the value for Y. Further, normally, the intended interpretation of a machine-dependent constant in compiler code is the value for the target class associated with the code, that is, Z. For example, an occurrence of ADDRESS-UNITS-PER-CONS-CELL in a PRIME primop almost certainly refers to the PRIME value of this constant, that is, 4. In order to allow both possible interpretations, Y or Z, occurrences of target-dependent constants for which the intended interpretation is Z are embedded in a special 'target' construct. The default interpretation is for Y.

All occurrences of target-dependent constants for which the Y value is intended are folded to the Y value. Given that internal compiler constants are rebound when retargeting, this folding is essential in code which is to be run during compilation. The handling of target-dependent constants referring to Z depends on the target class associated with the code. If the constant is in fact constant over the target class then it is folded to that value. Otherwise, it is treated as a special variable and bound to the appropriate target dependent value at compiler runtime.

Similar problems apply in the analysis of target dependent types, such as FIXNUM. The compiler uses special parameterized forms of the type predicates and analyzers, and different forms depending on whether analysis for Y or Z is required. For example, TARGET-FIXNUMP determines whether a given literal is a FIXNUM for the target by accessing the target values of MOST-POSITIVE-FIXNUM and MOST-NEGATIVE-FIXNUM. Type information in the compiler is maintained in a normal form in which all target-dependent types are expanded into machine-independent forms, thus allowing the host SUBTYPEP predicate to be used, for example, when determining whether a type-specific form of an operator may be applied.

4.5 *Derived-constants*

Many macros and constants are not themselves basic, but are derived from others. For example, the macro

```
(defmacro shift-by-tag-position (x) `(ASH ,x (- tag-position))
```

is machine-dependent only in its reference to the machine-dependent constant TAG-POSITION; thus SHIFT-BY-TAG-POSITION would not have to be tailored to each machine, but could be in the common compiler-machine. To serve this purpose, we introduce the notion of a 'derived' constant, whose evaluation form is remembered and not actually evaluated until first access to the constant. DEF-DERIVED-CONSTANT is like the Common Lisp DEFCONSTANT, except that the evaluation of the form which defines the constant is delayed until first access (this is a bit like lazy evaluation). The compiler will fold any constant down at compile time: What goes into the compiled file is the final, compile-time constant—not the form from which it was derived.

In general, a derived constant must be re-derived when the compiler changes target and then only when the value of the derived constant is needed. The value of a derived constant is cached once computed, and the cache is invalidated when the target is changed.

As an example of the use of derived constants, in the bignum development [White 1986], LOG-2-BITS-PER-ABIGIT is derived as

```
(and (power-of-two-p bits-per-abigit)
     (1- (integer-length bits-per-abigit)))
```

where BITS-PER-ABIGIT is a target-dependent constant.

In this way, many hundreds of lines of code that would otherwise have to be part of a machine-specific module are kept in the common modules.

5. Performance Statistics

Since Lucid is involved in maintaining its system on numerous different machines, we are concerned with the relative size of, and with the developmental effort involved in, the machine-dependent modules. We are concerned not only with the running speed of the compiler's output, but also with the degree of effort to maintain multiple implementations of Lucid Common Lisp.

The source code for the machine independent part of the compiler is 65% of the total source code size, 5% is tag-scheme specific, and 30% is CPU-specific. The CPU-specific code consists primarily of many short descriptions of primops.

The size of the CPU-specific part of the compiler indicates the degree of tailoring that is possible for a specific target architecture, and this tailoring is responsible for the relatively high performance of the resulting compiled code. The following elapsed times are for the Puzzle benchmark [Gabriel 1985]. This benchmark exercises two-dimensional arrays, function calling, and FIXNUM arithmetic.

Lucid Lisp is shown to be within 17% of the C version of Puzzle running on the Sun 3—an MC68020; the Sun 2 is an MC68010-based machine, running at 10 megahertz, as are the other MC68010-based machines listed. The IFU is an **I**nstruction pre-**F**etch **U**nit, which pre-fetches instructions for the Symbolics 3600.

Puzzle (Elapsed Seconds)	
Implementation	Time
C (Sun 3)	4.5
Lucid Lisp (Sun 3)	5.4
Symbolics 3600 + IFU	11.0
Symbolics 3600	13.9
Vax Lisp (Vax 8600)	15.5
Lucid Lisp (Sun 2)	17.5
PSL (MC68010)	26.3
Franz Lisp (MC68010)	51.1

6. Conclusions

The dynamically retargetable compiler has been shown to be a means by which ease of compilation for a variety of Lisp implementations can be performed; a means of porting Lisp systems to a variety of computers; and a tool for producing high-quality, high-performance code for a variety of computers from a common source.

References

[**Brooks 1982a**] Brooks, R. A., Gabriel, R. P., Steele, G. L. *An Optimizing Compiler For Lexically Scoped Lisp*, Proceedings of the 1982 ACM Compiler Construction Conference, June, 1982.

[**Brooks 1982b**] Brooks, R. A., Gabriel, R. P., Steele, G. L. *S-1 Common Lisp Implementation*, Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, August 1982.

[**Foderaro 1982**] Foderaro, J. K., Sklower, K. L. **The FRANZ Lisp Manual**, University of California, Berkeley, Berkeley, California, April 1982.

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

[**Griss 1982**] Griss, Martin L, Benson, E. *PSL: A Portable Lisp System*, Proceedings of the 1982 ACM Symposium on Lisp and Functional Programming, August 1982.

[**Leverett 1980**] Leverett, B., Szymanski, T. G. *Chaining Span-Dependent Jump Instructions*, ACM Transactions on Programming Languages and Systems, Vol. 2, No. 3, July 1980, Pages 274-289.

[**Steele 1978**] Steele, Guy Lewis Jr., *RABBIT: A Compiler for SCHEME (A Study in Compiler Optimization)*, Technical Report 474, Massachusetts Institute of Technology Artificial Intelligence Laboratory, May, 1978.

[**White 1986**] White, Jon L., *Reconfigurable, Retargetable Bignums:A Case Study in Efficient, Portable Lisp System Building* Proceedings of the 1986 ACM Symposium on Lisp and Functional Programming, August 1986.