SP *a professional paper*

ALGORITHMIC COMPILATION OF PREDICATES

by

Jeffrey A. Barnett

16 February 1968

SYSTEM DEVELOPMENT CORPORATION
2500 COLORADO AVE.
SANTA MONICA
CALIFORNIA
90406

ABSTRACT

A special technique for the algorithmic compilation of predicates is used by the LISP 2 and COMETA compilers produced at System Development Corporation. This technique generates highly efficient code; in addition, it has the further advantages of being easy to implement and of providing increased power and flexibility to the processor. This paper describes that technique by defining a model language, machine, and compiler, and then demonstrating briefly how the technique operates.

INTRODUCTION

A special technique for the algorithmic compilation of predicates is used by the LISP 2 and COMETA compilers produced at System Development Corporation.* This technique generates highly efficient code; in addition, it has the further advantages of being easy to implement and of providing increased power and flexibility to the processor. This paper describes that technique by defining a model language, machine, and compiler, and then demonstrating briefly how the technique operates.

Predicates are defined as forms that conditionally place program control, rather than evaluating as a datum. They are most often used as the antecedents of conditional forms and for the evaluation of some Boolean-valued forms. The distinction between a predicate and a Boolean expression can best be shown by the following examples:

```
1. (SETQ B(AND X Y))
2. (COND ((AND X Y) R) ...)
```

In the first example, the AND form is used as a Boolean expression; B is set to T or NIL, depending on the evaluation of the form. In the second example, the AND form is used as a predicate; evaluating the AND form merely determines what is to be evaluated next. A copy of the value of (AND X Y) is not desired, only the action of placing program control.

Predicates can be further understood by considering some of the classes of evaluation. The evaluation of forms may be divided into disjoint classes. As an example, (PLUS A B) is in the *value* class. The value of A is added to the value of B, producing a third value. Forms normally used as statements, e.g., (GO L), produce no value and therefore are in the *novalue* class. The *predicate* class consists of a group of forms for which the evaluation produces, like statements, no value. *Predicate*-class forms, however, place program control in some conditional manner. They should also be differentiated from *locative*-class forms, which are found on the left side of assignment expressions.

MODEL LANGUAGE

This paper deals with the compilation of AND, OR, and NOT forms used as predicates. Boolean constants and variables are also included in the language. The syntax of the model language is given below in BNF:

*This work was supported in part by the Advanced Research Projects Agency.

```
<predicate>      =  <negation>|<conjunction>|<variable>|<boolean>
<negation>       =  (NOT <predicate>)
<conjunction>    =  <union>|<intersection>
<union>          =  (OR <pstring>)
<intersection>   =  (AND <pstring>)
<pstring>        =  <empty>|<predicate> <pstring>
<variable>       =  <identifier>
<boolean>        =  T|NIL
```

Syntactically, predicates have the same form as Boolean expressions. Semantically, the evaluation of predicates may be defined in terms of the evaluation of the corresponding Boolean expression. If the value of the corresponding Boolean expression is NIL, the value of the predicate is *FALSE*; otherwise, the value of the predicate is *TRUE*. Note that T and NIL are data. True and false, on the other hand, are not data; they merely indicate the placement of program control.

The boolean T is the constant for true evaluation; the boolean NIL is the constant for false evaluation.

A boolean variable is an identifier--that is, any atom that would be a legal LISP variable name. If the value of the variable is NIL, the evaluation is false; it is true otherwise.

An intersection has the value true if none of its arguments evaluate to false; it is false otherwise. The evaluation need only proceed far enough to determine the value. If any of the arguments are false, the value of the intersection is false and the remaining arguments need not be evaluated.

Similarly for union, the value is true if any of the arguments evaluates true. The evaluation may cease when the first true argument is encountered.

A negation evaluates to true if its argument evaluates to false; it evaluates false otherwise.

MODEL MACHINE

The code generated by the example compiler will operate on any machine with an accumulator and directly addressable variables and instructions. Only four order codes are used:

1. LOAD (load accumulator). The address portion of a LOAD instruction is a variable. The action is to copy the value of the variable into the accumulator.

unnecessary - not used

2. BUC (branch unconditional). The address portion of a BUC instruction is a program label. The BUC transfers control to that label.

3. BOF (branch on false). The address portion of a BOF instruction is a program label. The BOF transfers program control to that label if the contents of the accumulator is NIL; it "falls through" otherwise.

4. BOT (branch on true). The address portion of a BOT instruction is a program label. If the contents of the accumulator is NIL, the instruction "falls through." Otherwise, control is transferred to the label.

MODEL COMPILER

Several trivial simplifications are first done by the example compiler (see Appendix A):

1. (OR) $\rightarrow$ NIL
2. (AND) $\rightarrow$ T
3. (OR P_1) $\rightarrow$ P_1
4. (AND P_1) $\rightarrow$ P_1
5. (OR P_1 ... P_N) $\rightarrow$ (OR P_1 (OR P_2 ... P_N)) $N \geq 2$
6. (AND P_1 ... P_N) $\rightarrow$ (AND P_1 (AND P_2 ... P_N)) $N \geq 2$

where P_1 ... P_N are arbitrary predicates.

The transformations are all consistent with the evaluation rules for union and intersection given above.

Several special variables are used in the example compiler. FGO holds a label to which program control is transferred if the predicate evaluation is false. TGO holds a label to which program control is transferred if the predicate evaluation is true. Either TGO or FGO, but not both, may be NIL. This indicates that program control of the generated code should "fall through" for the appropriate evaluation. EXP holds the symbolic form that is being compiled. CLASS is set to indicate the class of evaluation of a compiled form. In the example compiler, PRED (predicate) and VALUE are the only possible classes.

LISTING holds a list of symbolic machine instructions and labels. This list is built in reverse order and needs to be reversed before assembly.

The main routine of the predicate compiler is COMPRED, a function of three arguments: a form to compile as a predicate; a label to transfer to on true evaluation; and a label to transfer to on false evaluation. The form is compiled; if the class of evaluation is not PRED, the appropriate test and branch instructions are added to the listing.

COMNOT compiles negations. The imbedded predicate is compiled with the branch on true and branch on false labels reversed. An alternative to label reversion is exchanging the use of the opcodes BOF and BOT. However, the latter method is not easy and takes much extra work to produce code of the same quality.

COMMAND and COMOR compile intersections and unions, respectively, and use the subsidiary function COMBOOL.

COMBOOL has one argument, which is T for compilation of intersections and NIL for compilation of unions. The six simplification transformations listed above are used by COMBOOL. The actual compilation is done using only unions and intersections of two arguments. The first argument "falls through" on true or false for intersection and union, respectively. If the first argument is true for a union or false for an intersection, the evaluation ceases and program control is transferred to either TGO or FGO. If the appropriate label is NIL, indicating a "fall-through" of the entire conjunction, a label must be generated and placed at the end of the code produced for the conjunction. This label may be used for either TGO or FGO while compiling the first argument of the conjunction. The second argument is compiled with the same TGO and FGO as is used for the entire conjunction.

COMPILE is the master switch which directs all recursion within the compilation process. The argument to COMPILE is a form to be compiled. The appropriate function COMMAND, COMOR, COMNOT or COMATM is evoked for intersections, unions, negations, or Boolean constants and variables, respectively.

COMATM compiles atomic form. If the atom is a Boolean constant, an appropriate unconditional branch, BUC, is added to the LISTING. If a "fall-through" is indicated, no code is generated. For Boolean constants, CLASS is set to PRED. For variable forms, the code necessary to copy the value of the variable in the accumulator is generated and CLASS is set to VALUE.

ATTACH and ATTACHI are functions used to tack instructions and labels onto LISTING.

PREDICATE TRANSFORMATION

The example compiler has an interesting property: forms that are equivalent under DeMorgan transformations produce identical code sequences. The DeMorgan transformations are:

1. (NOT(AND P_1 P_2)) = (OR(NOT P_1)(NOT P_2))
2. (NOT(OR P_1 P_2)) = (AND(NOT P_1)(NOT P_2))

where P_1 and P_2 are any predicates.

A sketch of the proof is given here. Note that it is necessary to consider only conjunctions of two arguments, since conjunctions of more (or less) than two arguments can be simplified according to the six simplification rules given above. There are two cases to be considered: when neither the original TGO nor FGO is NIL, and when one or the other is. Also, it is sufficient to show that if either side of the DeMorgan equality is compiled, COMPRED will receive P_1 and P_2 with the same TGO and FGO.

Consider the first DeMorgan transformation with an initial TGO of TL and an initial FGO of FL (see Table 1).

Table 1. Arguments of COMPRED for First Transformation With Both TGO and FGO Non-Nil

Predicate	TGO	FGO	Predicate	TGO	FGO
(NOT(AND P_1 P_2))	TL	FL	(OR(NOT P_1)(NOT P_2))	TL	FL
(AND P_1 P_2)	FL	TL	(NOT P_1)	TL	NIL
P_1	NIL	TL	P_1	NIL	TL
P_2	FL	TL	(NOT P_2)	TL	FL
			P_2	FL	TL

Note that in Table 1, P_1 and P_2 have been compiled by COMPRED for both forms with identical TGO's and FGO's; they are compiled in the same order, and therefore must generate precisely the same code.

Consider next the case (as shown in Table 2) where TGO is originally NIL. (*genlab* is a generated label placed at the end of the code produced for the entire compilation.)

Table 2. Arguments of COMPRED for First Transformation with TGO NIL

Predicate	TGO	FGO	Predicate	TGO	FGO
(NOT(AND P_1 P_2))	NIL	FL	(OR(NOT P_1)(NOT P_2))	NIL	FL
(AND P_1 P_2)	FL	NIL	(NOT P_1)	genlab	NIL
P_1	NIL	genlab	P_1	NIL	genlab
P_2	FL	NIL	(NOT P_2)	NIL	FL
			P_2	FL	NIL

Again, the TGO's and FGO's for the compilation of P_1 and P_2 are identical.

Similarly, the case where FGO is NIL is shown in Table 3. Once again, the TGO's and FGO's are identical for P_1 and P_2 for the equivalent forms.

Table 3. Arguments of COMPRED for First Transformation with FGO NIL

Predicate	TGO	FGO	Predicate	TGO	FGO
(NOT(AND P_1 P_2))	TL	NIL	(OR(NOT P_1)(NOT P_2))	TL	NIL
(AND P_1 P_2)	NIL	TL	(NOT P_1)	TL	NIL
P_1	NIL	TL	P_1	NIL	TL
P_2	NIL	TL	(NOT P_2)	TL	NIL
			P_2	NIL	TL

A similar argument applies to the second DeMorgan transformation:

$$(\text{NOT}(\text{OR } P_1\ P_2)) \quad = \quad (\text{AND}(\text{NOT } P_1)(\text{NOT } P_2))$$

but is not shown here for the sake of brevity. Therefore, the code produced by the example compiler is invariant under DeMorgan transformations. Further, it is self-evident that the code generated for either the double negation, (NOT(NOT P_1)), or P_1 is identical (where P_1 is any predicate). Also, it is evident that the code sequences produced by conjunctions equivalent under an "arbitrary-nesting" transformation are identical. For example, the code produced for

$$(\text{OR } P_1\ P_2\ P_3\ P_4)$$

is identical to the code produced for

$$(\text{OR}(\text{OR } P_1(\text{OR } P_2\ P_3))(\text{OR}(\text{OR}(\text{OR } P_4))))$$

CONCLUSION

Using a model language, machine, and compiler, a technique for the compilation of predicates has been described. It has been shown that predicates which are identical under DeMorgan transformations, double negation and arbitrary nesting produce equivalent code when compiled. Several advantages of using this technique are: (1) it can be implemented easily in many compilers; (2) it has been found to interact naturally with components of complex processors (such as LISP); (3) it allows the programmer to employ any one of several source language forms (that may be "natural" to him), and yet produces identical code; (4) the code produced (see examples in Appendix B), is highly efficient. Algorithmic compilation of predicates is presently being used in LISP 2 and COMETA compilers at System Development Corporation. The techniques employed, however, are applicable to the compilation of predicates in any high-level programming language.

APPENDIX A

Listing of Example Compiler

```
SPECIAL((CLASS EXP LISTING TGO FGO))
DEFINE(((COMPRED (LAMBDA (X TGO FGO)
    (PROG NIL (COMPILE X)
     (COND ((NOT (EQ CLASS (QUOTE PRED)))
       (PROG NIL (COND (TGO (ATTACHI (QUOTE BOT) TGO)))
        (COND (FGO (ATTACHI (QUOTE BOF) FGO)))
        (SETQ CLASS (QUOTE PRED))))))))
  (COMNOT (LAMBDA NIL (COMPRED (CADR EXP) FGO TGO)))
  (COMAND (LAMBDA NIL (COMBOOL T)))
  (COMOR (LAMBDA NIL (COMBOOL NIL)))
  (COMBOOL (LAMBDA (B)
    (COND ((NULL (CDR EXP)) (COMPRED B TGO FGO))
     ((NULL (CDDR EXP)) (COMPRED (CADR EXP) TGO FGO))
     (T ((LAMBDA (LAB)
        (PROG NIL (COMPRED (CADR EXP)
          (COND (B NIL) (TGO TGO) (T LAB))
          (COND ((NOT B) NIL) (FGO FGO) (T LAB)))
         (COMPRED (CONS (CAR EXP) (CDDR EXP)) TGO FGO)
         (ATTACH LAB))) (GENSYM))))))
  (COMPILE (LAMBDA (EXP)
    (COND ((ATOM EXP) (COMATM))
     (T (SELECT (CAR EXP)
       ((QUOTE AND) (COMAND))
       ((QUOTE OR) (COMOR)) ((QUOTE NOT) (COMNOT)) (ERROR))))))
  (COMATM (LAMBDA NIL (SELECT EXP (NIL (PROG NIL (COND (FGO (ATTACHI
         (QUOTE BUC) FGO))) (SETQ CLASS (QUOTE PRED))))
     (T (PROG NIL (COND (TGO (ATTACHI (QUOTE BUC) TGO)))
       (SETQ CLASS (QUOTE PRED))))
     (PROG NIL (ATTACHI (QUOTE LOAD) EXP)
      (SETQ CLASS (QUOTE VALUE))))))
  (ATTACHI (LAMBDA (OP ADR) (ATTACH (LIST OP ADR))))
  (ATTACH (LAMBDA (I) (SETQ LISTING (CONS I LISTING))))))
```

APPENDIX B

Code Produced By Example Compiler

The function DRIVER is an example supervisor for interacting with the model compiler.

```
DEFINE (((DRIVER (LAMBDA NIL (PROG (EXP TGO FGO CLASS)
             L (PROG (LISTING)
              (COMPRED (READ) (QUOTE TRUE) (QUOTE FALSE))
              (TEREAD) (PRINT  (REVERSE LISTING)) (TERPRI)) (GO L))))))
```

DRIVER was used in conjunction with the model compiler to generate the code shown in the examples below.

<u>EXAMPLE 1</u>: Equivalence of Code Produced Under Arbitrary Nesting

```
(OR (OR (OR P1 P2) P3)                  (OR P1 P2 P3 P4)
 (OR (OR (OR (OR P4)))))

      ((LOAD P1)                          ((LOAD P1)
       (BOT TRUE)                          (BOT TRUE)
       (LOAD P2)                           (LOAD P2)
       (BOT TRUE)                          (BOT TRUE)
   GEN3(LOAD P3)                           (LOAD P3)
       (BOT TRUE)                          (BOT TRUE)
   GEN2(LOAD P4)                           (LOAD P4)
       (BOT TRUE)                          (BOT TRUE)
       (BOF FALSE)                         (BOF FALSE)
   GEN1)                                GEN3 GEN2 GEN1)
```

Note that the spuriously generated labels (GEN1, GEN2, and GEN3) may appear in different places for the equivalent input forms. However, if the labels were referenced, they would appear at the same relative location in the code sequences produced.

APPENDIX B (Cont.)

EXAMPLE 2: Equivalence of Code Produced Under DeMorgan Transformation and Double Negation

```
(OR (AND (NOT P1)(NOT P2)(NOT P3))(NOT P4))
(NOT (AND (OR P1 P2 P3) P4))

            ((LOAD P1)
             (BOT GEN2)
             (LOAD P2)
             (BOT GEN3)
             (LOAD P3)
             (BOF TRUE)
         GEN3 GEN2
             (LOAD P4)
             (BOT FALSE)
             (BOF TRUE)
         GEN1)
```

EXAMPLE 3: Code Produced For a Complicated Predicate

```
(AND (NOT (OR P1 P2 P3))
 (OR P4 (NOT P5)(AND P6 (NOT P7)))
 (OR P8 (AND P9 P10)))

         ((LOAD P1)
          (BOT FALSE)
          (LOAD P2)
          (BOT FALSE)
          (LOAD P3)
          (BOT FALSE)
      GEN3 GEN2
          (LOAD P4)
          (BOT GEN5)
          (LOAD P5)
          (BOF GEN6)
          (LOAD P6)
          (BOF FALSE)
          (LOAD P7)
          (BOT FALSE)
      GEN7 GEN6 GEN5
          (LOAD P8)
          (BOT TRUE)
          (LOAD P9)
          (BOF FALSE)
          (LOAD P10)
          (BOT TRUE)
          (BOF FALSE)
      GEN9 GEN8 GEN4 GEN1)
```