CODE IMPROVEMENT VIA LAZY EVALUATION

David R. HANSON
Department of Computer Science, The University of Arizona, Tucson, AZ 85721, U.S.A.

Received 6 May 1980; revised version received 26 August 1980

Code generation, lazy evaluation, peephole optimization

1. Introduction

Many computers have 'polymorphic' instructions that can be used with different forms of operands. Different forms typically correspond to different addressing 'modes'. Examples of common operand forms are direct operand, in which the address of the operand is given, indexed operand, in which the location of the operand is indicated by an offset and the contents of a base register, and immediate operand, in which the address *is* the operand. These kinds of operand variations appear on several commonly used computers – the DEC-10 and the PDP-11 series are prime examples – as well as an increasing number of emerging microprocessors.

Operand variations permit more compact (and hence faster) code for some constructs. For example, consider the expression

a = b + c.

Taking the DEC-10 as the target computer, naive code for this expression is

```
move   1,b:   load b into register 1
move   2,c:   load c into register 2
add    1,2;   compute sum in register 1
movem  1,a:   store sum in a.
```

This kind of code is simple to generate and illustrates the typical approach used in naive code generators of placing operands in registers because almost all instructions can operate on registers. (Similar comments apply to the PDP-11, for example.) The add instruction can operate on either registers or memory, however, so that better code is

```
move   1,b:   load b into register 1
add    1,c:   compute sum in register 1
movem  1,a:   store sum in a.
```

A similar situation arises when constants are involved in an expression. In that case, the use of immediate operands can result in significant savings. Consider, for example, the expression

a = b + 5.

Naive code generation results in

```
move   1,b:   load b into register 1
move   2, [5]: load 5 into register 2
add    1,2:   compute sum in register 1
movem  1,a:   store sum in a
```

where the brackets indicate the address of a word containing what they enclose. This code occupies 5 words on the DEC-10. Both the move and add instructions can be used with immediate operands. Thus better code is

```
move   1,b:   load b into register 1
movei  2,5:   load 5 into register 2
add    1,2:   compute sum in register 1
movem  1,a:   store sum in a
```

and even better code is

```
move   1,b:   load b into register 1
addi   1,5:   compute sum in register 1
movem  1,a;   store sum in a.
```

Code improvement of the type described above is

usually considered 'peephole' optimization [1,11] and, in fact, can be performed routinely using more recent generalized approaches to peephole optimization [2,4]. The disadvantage of relying solely on a separate optimization phase for such code improvement is that the optimization phase is invariably the last part of the compiler to be done, and, in the interim, poor code may discourage use.

The remainder of this note describes how 'lazy evaluation' can be used to improve the code produced by naive generators. Experimental results suggest that its use can reduce the size of the generated code for expressions by 15 to 20%. For example, it produces the short three-instruction sequence for the example given above. The attractive aspect of the technique is that it is easy to implement, thereby making it possible to get reasonable code out of a naive code generator with very little additional effort.

2. Lazy evaluation

Lazy evaluation is a technique often used in the evaluation of programming language constructs [5,6, 12,13]. Its basic idea, often presented in a Lisp framework, is to defer actual evaluation of an expression until the value is really needed, e.g. for output. In this scheme, evaluation of an expression results in a 'suspension' or 'closure' that represents the expression. Only when an attempt is made to use the value of the suspension is the expression actually evaluated. Among other advantages, lazy evaluation permits the representation of infinite structures and, in some cases, results in more efficient computation.

The concepts underlying lazy evaluation apply equally well to code generation. The basic idea is to defer generating code for operands until an operand is really needed, e.g. in an instruction.

Code generation for the 'leaves' of an expression tree, e.g. references to constants and variables, amounts to building a suspension that represents the reference. Only when the value of the suspension is needed in an instruction is code generated for referencing the operand, *if necessary*. The key point is that, since code for referencing the operand has not been generated, the addressing modes that best suit the operands can be used when the instruction is produced.

To be effective, it is necessary to express basic machine operations as 'generalized' instructions that accept any of the possible operand forms. The generation of the specific form is deferred until it is output, since the form depends on the operands. Moreover, generation of the specific instruction is localized to the final output routine, which *is* the lazy evaluator, and is therefore performed for all generated code sequences.

```
genop(oper, s1, s2) |
  case kind of s1 suspension |
    'register':
      LOP = 'register number'
    'constant':
      if constant is small and oper can accept immediate operand then
        LOP = 'immediate constant'
      else |
        output load for constant
        LOP = 'register number'
        |
    'address':
      if oper cannot accept address as first operand then |
        output load for s1
        LOP = 'register number'
        |
      else
        LOP = 'address'
    |
    case kind of s2 suspension |
    'register':
      ROP = 'register number'
    'constant':
      if constant is small and oper can accept immediate operand then
        ROP = 'immediate constant'
      else |
        output load for constant
        ROP = 'register number'
        |
    'address':
      if oper cannot accept address as second operand then |
        output load for s2
        ROP = 'register number'
        |
      else
        ROP = 'address'
    |
  output 'oper LOP, ROP'
  return suspension representing result, e.g. ['register', LOP]
```

Fig. 1.

To clarify the mechanics of the technique, consider a situation in which there are three kinds of operands: registers, constants, and addresses. The heart of the code generation routine is *genop(oper, s1, s2)*, which generates an instruction sequence that carries out the application of *oper* on the suspensions *s1* and *s2*. The operation of *genop* is outlined in Fig. 1 in which the suspension construction is indicated by brackets. Assuming the code generator is driven by a tree representing the expression, the code for

a = b + c

is generated as follows. Working from the leaves of the

tree up toward the root, suspensions representing b and c are produced:

s1 = ['*address*', b];
s2 = ['*address*', c]:

genop is called to generate code for the addition:

s3 = *genop*(add, *s1*, *s2*).

which, in the process, emits code to load b

move 1,b: load b into register 1

followed by the addition:

add 1,c: compute b + c in register 1.

At this point, *genop* returns a suspension representing its result:

s3 = ['*register*', *1*].

Finally, similar actions for the assignment produce

movem 1,a: store sum in a.

Applying a similar process to the expression

a = b – 5

yields the suspensions

s1 = ['*address*', b]
s2 = ['*constant*', 5]

followed by the emission of

```
move  1,b:  load b into register 1
subi  1,5:  compute b – 5 in register 1
movem 1,a:  store in a.
```

The suspension for the final result is *s3*, given above. If the constant in the previous example had been too large to use as an immediate operand,

sub 1,[5]

would have been generated in place of the subi.

By generalizing the instruction set even further, lazy evaluation can simplify other aspects of code generation. For example, if type information is placed in suspensions, *genop* can use that information to select the appropriate instruction. Continuing with the initial example used above, suppose that b and c are floating point variables. Calling *genop*(add, *s1*, *s2*) specifies a generic addition that, by inspecting the type information in *s1* and *s2*, results in the specific instruction for floating point addition:

```
move 1,b:  load b into register 1
fadr 1,c:  compute floating point sum in register 1.
```

Mixed mode operations can be handled in a similar fashion. Suppose b is an integer and c is a floating point requiring that the value of b be converted to floating point prior to the addition. The need for the conversion can be detected in *genop* by inspecting the operation and the types of the operands resulting in the following code:

```
fltr 1,b:  load and convert b
fadr 1,c:  compute floating point sum in register 1.
```

Note that if b and c are integers and a is floating point, the resulting code is

```
move  1,b:  load b into register 1
add   1,c:  compute sum in register 1
fltr  1,1:  convert sum to floating point
movem 1,a:  store in a.
```

The type of the suspension returned after the addition is integer, which triggers conversion to floating point prior to the assignment to a.

3. Discussion

In addition to better code, the payoff from the lazy evaluation approach is that it is easy to implement. It has been used in the code generator for the Y programming language [6], which is a simple systems programming language derived from Ratfor [8,9] and C [10]. The first version of the Y code generator was the typical naive code generator. Revision to use lazy evaluation took about 2 days and yielded a 15 to 20% reduction in code size for expressions. (Programming time would have been even less had the lazy evaluation technique been used initially.) Programs typically ran within 10% of the speed of equivalent Fortran programs, which were compiled using the standard DEC-10 Fortran compiler [3]. The technique fits very well into the naive approach to code generation, making its inclusion negligible compared to its benefits.

It should be made clear what lazy evaluation, in the form presented here, is *not*. It cannot, for example, detect short instruction sequences that can be replaced

by shorter sequences, which can be done through peephole optimization [2]. For example, the expression

```
a = a + 1
```

produces

```
move  1,a;  load a into register 1
addi  1,1;  increment it
movem 1,a;  store sum in a
```

which can be replaced in a peephole optimizer by

```
aos  1,a;  increment a.
```

The reason lazy evaluation cannot be used to detect such sequences is that it deals only with operands of a single instruction, not with sequences of instructions.

Lazy evaluation, in its simplest form, is also not a register allocation scheme. Its primary purpose is to defer putting operands in registers instead of remembering what is in registers. It is more like register *assignment* rather than register allocation [1]. As such, it cannot, for instance, be easily used to detect redundant loads and stores. For example, the expressions

```
a = b + c
b = a + 5
```

produce

```
move  1,b;  load b into register 1
add   1,c;  compute b + c in register 1
movem 1,a;  store sum in a
move  1,a;  load a into register 1
addi  1,5;  compute a + 5 in register 1
movem 1,b;  store sum in b
```

which contains a redundant load of a.

In analyzing the behavior of the initial lazy evaluation implementation, it was observed that suspensions represent not only what needs to be done, but can also record the current state of the machine. Consider again the expression

```
a = b + c.
```

The references to b and c result in suspensions as described in the previous section, and calling *genop* to generate the addition causes interrogation of the suspension for b, which causes emission of

```
move 1,b:  load b into register 1.
```

At this point, instead of discarding the suspension for b, it can be modified to reflect the fact that b is in register 1. After the rest of the code is emitted:

```
add   1,c:  compute b + c in register 1
movem 1,a:  store in a
```

the suspension for a can be modified to indicate that a is now in register 1 and the suspension for b is restored to its original state. Current efforts are directed toward handling the suspensions in this fashion and viewing them as a cache. Preliminary results suggest that this generalization of the lazy evaluation technique might permit it to handle some kinds of peephole optimizations as well as register allocation.

Acknowledgment

This work has benefited greatly from discussions with Chris Fraser, Jack Davidson, Peter Downey, and Ravi Sethi. This work was supported by the National Science Foundation under Grant MCS78-02545.

References

[1] A.V. Aho and J.D. Ullman, Principles of Compiler Design (Addison-Wesley, Reading, MA, 1977).
[2] J.W. Davidson and C.W. Fraser, A retargetable peephole optimizer and its application to code generation, ACM Trans. Progr. Languages and Systems 2 (2) (1980).
[3] Fortran-10 Language Reference Manual, Pub. DEC-10-LFORA-B-D, 2nd ed. (Digital Equipment Corp., Maynard, MA, 1974).
[4] C.W. Fraser, A compact, machine-independent peephole optimizer, Conference Records Sixth ACM Annual Symposium on the Principles of Programming Languages, San Antonio (Jan. 1979) 1–6.
[5] D. Friedman and D.S. Wise, CONS should not evaluate its arguments, in: S. Michaelson and R. Milner, Eds., Automata, Languages and Programming (Edinburgh Univ. Press, Edinburgh, 1976) 257–284.
[6] D.R. Hanson, The Y programming language, SIGPLAN Notices, to appear.
[7] P. Henderson and J.H. Morris, A lazy evaluator, Conference Records Third ACM Annual Symposium on the Principles of Programming Languages, Atlanta (Jan. 1976) 95–103.
[8] B.W. Kernighan, Ratfor – a preprocessor for a rational Fortran, Software – Practice and Experience 5 (4) (1975) 396–406.
[9] B.W. Kernighan and P.J. Plauger, Software Tools (Addison-Wesley, Reading, MA, 1976).

[10] B.W. Kernighan and D.M. Ritchie, The C Programming Language (Prentice-Hall, Englewood Cliffs, NJ, 1978).
[11] W.M. McKeeman, Peephole optimization, Comm. ACM 8 (7) (1965) 443–444.
[12] J. Vuillemin, Correct and optimal implementations of recursion in a simple programming language, J. Comput. System Sci. 9 (3) (1974).
[13] C. Wadsworth, Semantics and pragmatics of the Lambda-calculus, Ph. D. Dissertation, Oxford (1971).