Information Processing Letters 24 (1987) 391-395
North-Holland

6 April 1987

OPTIMIZATION OF ARGUMENT EVALUATION ORDER *

Christopher W. FRASER

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

David R. HANSON

Department of Computer Science, Princeton University, Princeton, NJ 08544, U.S.A.

Communicated by G.R. Andrews
Received 9 May 1986
Revised 30 June 1986

Keywords: Peephole optimization, expression evaluation, code generation

1. Introduction

In many languages, the order of evaluation of arguments is undefined or only partially defined. For example, in the C expression [4]

```
f(g( ), h( ))
```

either g or h will be invoked first, depending on the implementation. Similar situations arise in PASCAL and ADA [1].

Incomplete definition of evaluation order may lead to erroneous programs or unnecessary verbosity. For example, to insure that g is evaluated first in the expression above, the programmer must write

```
t = g( );
f(t, h( ));
```

Even apparently innocuous expressions can yield incorrect results. For example, in f(a, g( )), if a is a global variable that is modified by g, the value of a passed to f depends on the order of evaluation of the arguments. If the arguments are evaluated left-to-right, the value a before the call to g is passed; if the order is right-to-left, the value of a after calling g is passed. Program checkers that identify nonportable and error-prone constructs, such as lint [2], often miss these kinds of constructs.

While some of these examples can be critized as poor programming style (e.g., g modifying a), a well-defined evaluation order would at least give a precise interpretation for all expressions, obviate the need for useless temporary variables, and make bugs independent of implementation details.

A justification for leaving the evaluation order undefined is efficiency. The freedom to evaluate expressions in any order permits the implementor to tailor evaluation order to the architectural properties of the target computer, such as the direction of stack growth [3]. The remainder of this article describes a simple optimization that permits the evaluation order of arguments to be defined without sacrificing code quality for most calls. Calls with arguments that do not depend on the evaluation order are optimized to best suit the architecture of the target computer. For calls with arguments that *do* depend on the order of evaluation, the code is not as good, but it ensures a well-defined order of evaluation. Since most calls fall into the former category, the overall reduction in code quality is slight for most programs.

* This work was supported in part by the National Science Foundation under Grants MCS-8302398 and DCR-8320257.

The purpose of most optimizations is to get better object code from the same source code. The purpose of this optimization is to get the same object code from clearer source code. Thus, most optimizations are judged by quantitative improvements in object code, but this one must be judged by qualitative improvements in source code, such as the removal of hidden machine-dependencies.

2. Argument evaluation

In most languages with recursive functions, evaluation of call $f(e_1, \ldots, e_n)$ involves some combination of allocation of stack space for the n arguments, evaluation of the argument expressions, e_i, storing the resulting argument values into the allocated space, transferring to the entry point of the function, and deallocation of the argument space. The precise details of these steps depend mainly on the order of evaluation of the arguments, the direction of stack growth, and the layout of activation records, or *frames*, on the stack.

Details of stack growth are generally fixed by the architecture of the target machine. Frame layout is often dictated by constraints on based addressing since arguments are usually accessed by an offset from a dedicated base register, which is often called the *frame* or *argument* pointer. For example, on the IBM 370, such offsets must be nonnegative. Constraints concerning the allocation and addressing of local variables may also influence frame layout.

Frame layout can also be influenced by language features and compiler implementation considerations. For example, in languages that permit functions with a variable number of arguments, frames should be designed so that offsets to arguments are independent of the number of actual arguments. This criterion also simplifies code generation. Implementation of tail recursion can also impact frame layout [6].

In languages that do not specify the order of argument evaluation, the order that yields the most efficient calling sequence and argument addressing is typically used. Let a_i denote the location of the *ith* argument, $e_i \rightarrow a_i$ denote the evaluation of the *ith* argument and assignment of the resulting value to a_i, and *allocate* k denote the allocation of stack space for k arguments. Using this notation, a call $f(e_1, \ldots, e_n)$ in which the arguments are evaluated left-to-right might be expressed as

allocate 1
$e_1 \rightarrow a_1$
$\vdots$
allocate 1
$e_n \rightarrow a_n$
call f
deallocate n

where *call* f denotes the transfer to f, including establishing the return address, and *deallocate* deallocates the space allocated by *allocate*. Many machines have a 'push' instruction, which combines the effect of *allocate* and $e_i \rightarrow a_i$, i.e.,

$$\begin{matrix} \textit{allocate}\ 1 \\ e_i \rightarrow a_i \end{matrix} \Rightarrow \textit{push}\ e_i$$

But *push* can be used only if the resulting order of the arguments agrees with the frame layout. The order of argument evaluation is usually changed to meet this constraint.

For example, Fig. 1 shows the frame layout for the VAX. Frames are constructed in part by the call instruction. The stack grows 'down' toward

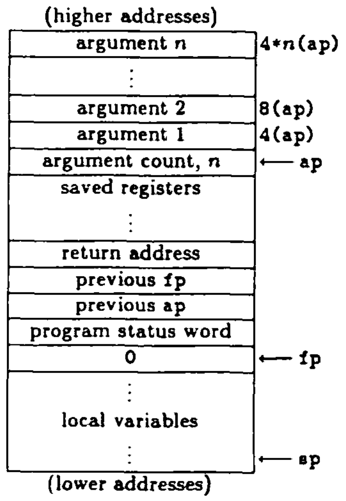

Fig. 1. Frame layout for the VAX.

lower addresses, and arguments are addressed by offsets from the argument pointer, ap. In order for the argument offsets to be independent of the number of arguments, the arguments must be pushed in the order n, n − 1, ..., 1, which is accomplished by the following calling sequence:

```
push e_n
   :
push e_1
call f
deallocate n
```

Because this sequence is best for the VAX, most VAX C compilers evaluate arguments from right to left. Other frame layouts, stack growth directions, and instruction sets induce the opposite order.

It is possible to specify the order of argument evaluation and use a calling sequence that is nearly as good as that given above by evaluating the arguments, saving the resulting values in registers, and pushing the registers in the correct order. This sequence on the VAX, for example, is

```
e_1 → r_1
   :
e_n → r_n
push r_n
   :
push r_1
call f
deallocate n
```

where r_i denotes a register. The disadvantage of this approach is that it may require n − 1 registers in addition to those necessary to evaluate the arguments, and registers may be scarce. Also, there may not be enough registers to evaluate the later arguments without using temporaries, resulting in poor code.

3. Optimizing argument evaluation

It is possible to resolve the dilemma between specifying argument evaluation order and generating efficient calling sequences by using a simple optimization. The resulting code quality suffers only for calls with arguments whose values appear to depend on the order of evaluation. In this case, the code is as good as if the programmer had changed the source code to establish the evaluation order explicitly. Suppose that the language specifies left-to-right argument evaluation. The calling sequence for a call $f(e_1, \ldots, e_n)$ is

```
allocate n
e_1 → a_1
   :
e_n → a_n
call f
deallocate n
```

For machines on which the desired order of *push* operations is 1, 2, ..., n, the optimizations

```
allocate k    ⇒   push e_i
e_i → a_i         allocate k − 1
allocate 0    ⇒   —
```

combine to convert the calling sequence above to one that uses *push* operations.

For machines on which the desired order of *push* operations is n, n − 1, ..., 1, the appropriate optimizations are

```
allocate k    ⇒   push e_k
e_k → a_k         allocate k − 1
allocate 0    ⇒   —
```

but these cannot be applied to the calling sequence above becauses the $e_i \rightarrow a_i$ are in the incorrect order. The correct order can be established by applying

```
e_i → a_i    ⇒   e_j → a_j
e_j → a_j        e_i → a_i
```

if $i < j$ and if the evaluation of e_j does not depend on the outcome of evaluating e_i, which is a common occurrence in many programs. The independence of the two expressions can be determined by conventional dataflow analysis or approximated by assuming that expressions are independent if and only if neither expression performs an assignment or a procedure call. It seems likely that the less expensive approximation will suffice in most cases.

Code quality suffers only when the evaluation of some of the arguments depends on other arguments, but some improvement is often possible. For example, if e_2 depends on e_1 in $f(e_1, e_2, e_3, e_4)$, the optimized calling sequence is

```
push e4
push e3
allocate 2
e1 → a1
e2 → a2
call f
deallocate n
```

The C expression f(a, a = b + 1, c, d) is an example of this specific dependency. Further improvements depend on architectural details, and their value must be weighed against their implementation cost and the expected importance of such calls. For example, if some register r is available, the preceding sequence can be improved to

```
push e4
push e3
e1 → r
push e2
push r
call f
deallocate n
```

The optimization involving *allocate* k above is the special case of i = j in the optimization

$$\begin{array}{l} \textit{allocate}\ i \\ e_j \to a_j \end{array} \Rightarrow \begin{array}{l} \textit{allocate}\ i-j \\ \textit{push}\ e_j \\ \textit{allocate}\ j-1 \end{array}$$

where $i \leqslant j$. This optimization allocates stack space for i arguments by distributing it between two allocations and a *push*. For example, if e_3 depends on e_2 in $f(e_1, e_2, e_3, e_4)$, the optimized sequence is

```
push e4
allocate 2
push e1
e2 → a2
e3 → a3
call f
deallocate n
```

Indiscriminate application of this last optimization can lengthen the sequence, making the code *worse*. It should be applied only when either i − j or j − 1 is 0.

This optimization can be implemented during code generation. Consider, for example, a compiler that generates code while walking an expression tree. The tree can be extended with nodes for *allocate* n, $e_i \to a_i$, and *push* a_i. The permutations specified by the optimization rules above can be implemented as the tree is traversed by editing the nodes as described in the rules and by changing the traversal order.

This optimization can also be implemented at the instruction level using traditional peephole optimization techniques [5]. On the VAX, for example, the correspondences between the abstract operations and VAX instructions are

```
allocate n      subl2  $4*n, sp
push ri         pushl  ri

call f
deallocate n    calls  $n, f
```

The subl2 instruction decrements sp so that it points to the location ultimately occupied by a_1, the first argument: thus, a_i refers to an offset of 4*(i − 1) from sp (see Fig. 1). Using these correspondences and the calling sequence given at the beginning of this section, the initial code for f(a, b + 1, c) is

```
subl2  $12, sp     allocate 3
movl   a, 0(sp)    e1 → a1
movl   b, r2       e2 → a2
addl2  $1, r2
movl   r2, 4(sp)
movl   c, 8(sp)    e3 → a3
calls  $3, f       call f: deallocate 3
```

The movl instructions that reference sp store argument values into the stack. In order to replace the subl2 and movl instructions by pushl instructions, the movl instructions must appear in the opposite order from that shown. If argument evaluation order was undefined, the desired order could have been generated initially. The desired order can be obtained with the optimizations

```
movl  any, i(sp)  ⇒  movl  any, j(sp)
movl  any, j(sp)     movl  any, i(sp)
```

where i < j (this condition ensures that the rules will not endlessly exchange a pair of movl instructions). *any* specifies any operand, and the two movl instructions commute.

```
movl  any, i(sp)  ⇒  inst
inst                 movl  any, i(sp)
```

where *inst* specifies any other instruction that commutes with the movl instruction, and

```
subl2  $k, sp  ⇒  inst
inst              subl2  $k, sp
```

where *inst* is any instruction that commutes with the subl2 instruction. Instructions x and y *commute* if the execution sequence x y is equivalent to y z. After two applications of the second and third optimizations, the above example becomes

```
movl   b, r2
addl2  $1, r2
subl2  $12, sp
movl   a, 0(sp)
movl   r2, 4(sp)
movl   c, 8(sp)
calls  $3, f
```

Three applications of the first optimization change the sequence to

```
movl   b, r2
addl2  $1, r2
subl2  $12, sp
movl   c, 8(sp)
movl   r2, 4(sp)
movl   a, 0(sp)
calls  $3, f
```

The subl2 instruction is now in the position necessary to apply the optimization

```
subl2  $4*n, sp
movl   any, 4*(n − 1)sp
                        ⇒  pushl  any
                           subl2  $4*(n − 1), sp
```

Three applications of this optimization followed by applying

```
subl2  $0, (sp) ⇒ —
```

yields

```
movl   b, r2
addl2  $1, r2
pushl  c
pushl  r2
pushl  a
calls  $3, f
```

References

[1] ADA, Reference Manual for the ADA Programming Language, Dept. of Defense, Washington, D.C., 1980.

[2] S.C. Johnson, Lint—a C program checker, Computing Science Tech. Rept. 65, Bell Laboratories, Murray Hill, NJ, December 1977.

[3] S.C. Johnson and D.M. Ritchie, The C language calling sequence, Computing Science Tech. Rept. 102, Bell Laboratories, Murray Hill, NJ, September 1981.

[4] B.W. Kernighan and D.M. Ritchie, The C Programming Language (Prentice-Hall, Englewood Cliffs, NJ, 1978).

[5] D.A. Lamb, Construction of a peephole optimizer, Software—Practice & Experience 11 (12) (1981) 639–648.

[6] G.L. Steele, Jr., Debunking the "expensive procedure call" myth, or procedure call implementations considered harmful, or LAMBDA: The ultimate GOTO, Proc. ACM Ann. Conf., Seattle, WA (1977) 153–162.