Programming Languages

B. Wegbreit Editor

Recursion Analysis for Compiler Optimization

Kenneth G. Walter
Case Western Reserve University

A relatively simple method for the detection of recursive use of procedures is presented for use in compiler optimization. Implementation considerations are discussed, and a modification of the algorithm is given to further improve optimization. This analysis can also be used to determine what possibile subset of values could be assumed by variables which can only take on a relatively small discrete set of values. The most common are parameters of variables assuming values of label, procedure, or Pascal's enumerated type.

Key Words and Phrases: recursion, compiler optimization

CR Categories: 4.12

Author's address: Computing and Information Sciences Department, Case Western Reserve University, Cleveland, OH 44106.

Introduction

Recursion analysis is a useful, though widely ignored, technique for an optimizing compiler. In a language which does not support recursion, like Fortran, it is very difficult for a programmer to detect inadvertent recursions. Recursion analysis can be used to provide diagnostics. In a language permitting recursion, like Algol 60, more efficient code can be generated for procedures known to be nonrecursive. In a language requiring that the declaration specify whether a procedure is recursive, mistakes can be detected.

One reason for the lack of recursion analysis in languages which permit recursion is the separate compliation of procedures. This can be remedied by the use of a suitable database or by making worst-case assumptions about external procedures. It will be assumed that all procedures are available for this analysis so that relations concerning them can be constructed.

Relations

To determine what procedures are recursive, a relation $\underset{\sim}{C}$ ("calls") can be constructed where P $\underset{\sim}{C}$ Q is true iff P calls Q. The transitive closure $\underset{\sim}{C}^{+}$ then determines whether any procedure can call itself and is therefore recursive. The first step in constructing $\underset{\sim}{C}$ is to determine the direct call relations. In each procedure P, if a procedure Q is called, then P $\underset{\sim}{C}$ Q is true. An example is given in Figure 1. In P, Q is called,

Fig. 1.
```
procedure P;
begin
    procedure Q;
    begin
      ⋮
      P;
      P;
    end;
  ⋮
  Q;
  ⋮
end;
```

thus P $\underset{\sim}{C}$ Q is true, and in Q, P is called, giving Q $\underset{\sim}{C}$ P. Computing $\underset{\sim}{C}^{+}$ yields P $\underset{\sim}{C}^{+}$ P and Q $\underset{\sim}{C}^{+}$ Q. Both P and Q are recursive.

A problem arises in recursion analysis where a procedure can appear as a parameter to a procedure (this includes call-by-name parameters). Any formal parameter which in implementation acts as a procedure shall be called a formal procedure. The formal procedure is included in the $\underline{C}$ relation, but it is also necessary to determine what values (actual procedures) the formal procedure can assume.

To determine this correspondence of formal procedures to actual parameters, another relation $\underline{P}$ ("has parameter value") is constructed. If some procedure P has formal parameter R, and a call $P(A)$ occurs in the program, then the relation $R\ \underline{P}\ A$ is true. At this point, notice that $\underline{P}$ is similar to $\underline{C}$ and can be interpreted as R calls A iff $R\ \underline{P}\ A$. The transitive closure $\underline{P}^+$ chains parameter passing through several procedures together. Using $\underline{P}$, it is then possible to determine that P calls P in Figure 2 by seeing that $P\ \underline{C}\ X$ and $X\ \underline{P}\ P$.

Fig. 2.

```
procedure A; begin ··· end;
procedure P(X);
begin
  ⋮
  X(A);
  ⋮
end;
  ⋮
  P(P);
```

The construction of $\underline{P}$ raises another problem. At a call $Q(X)$ where Q is a formal procedure, the formal parameter of Q is unknown. This makes it difficult to put something into the $\underline{P}$ relation to X. Figure 3 is an example where this occurs.

Fig. 3.

```
procedure A4(F4);
begin
  ⋮
  F4(sin);
  ⋮
end;
procedure A3(F3);
begin
  ⋮
  F3(3.14);
  ⋮
end;
A4(A3);
```

To solve this problem, dummy parameters (dummy procedures) are created for each formal procedure where needed. The $\underline{P}$ relation then holds between the dummy procedure and the actual parameter, i.e. "dummy" $\underline{P}\ X$. It is then necessary to identify formal procedures with their dummy parameters and actual procedures with their formal parameters. This is done with another relation $\underline{F}$ ("has formal parameter").

At this point, we will assume procedures have only one parameter of interest. If there were more than one parameter, calls on formal procedures would be processed as if all permutations of the actual parameters had occurred. In some circumstances this would mistakenly identify a procedure as recursive. The alternative is to build relations

$$F1, \cdots, Fn,$$

where Fi is the relation for the ith parameter position.

It is now possible to determine the correspondence between dummy and formal procedures. Using the example in Figure 3, $F3$ is in the $\underline{F}^t$ (transpose) relation to $A3$; $A3$ is in the $\underline{P}^t$ relation to $F4$; and $F4$ is in the $\underline{F}$ relation to $D4$ (its dummy procedure), giving $F3\ \underline{P}\ D4$. Computing $\underline{P}^+$ then allows the conclusion that a call of $F3$ can be a call on sin. This can be computed for all procedures by constructing the compound relation as

$$\underline{P} \leftarrow \underline{F}^t\underline{P}^{+t}\underline{F} \vee \underline{P}^+.$$

Figures 4 and 5 show that this is not a one-step process. A formal procedure may be called with a formal procedure as a parameter. To determine the correspondence for one dummy procedure, the correspondence with the other must first be determined. The algorithm runs as follows:

1. Initialize $\underline{P}$ and $\underline{F}$
2. $\underline{P} \leftarrow \underline{P}^+$
3. $\underline{P} \leftarrow \underline{F}^t\underline{P}^t\underline{F} \vee \underline{P}$
4. If $\underline{P}$ was changed in step 3 then go to step 2.

When $\underline{P}$ converges to the solution, it is combined into $\underline{C}$

5. $\underline{C} \leftarrow \underline{C} \vee \underline{P}$,

and $\underline{C}^+$ is computed.

Implementation

In Fortran, only subroutines or functions need to considered as they are the only objects called. In Algol

Fig. 4.

procedure $A4(F4)$;	**begin** ··· $F4(sin)$ ··· **end**;	⇒	$A4\ \underline{F}\ F4$,	$F4\ \underline{F}\ D4$,	$D4\ \underline{P}\ sin$
procedure $A3(F3)$;	**begin** ··· $A4(F3)$ ··· **end**;	⇒	$A3\ \underline{F}\ F3$,	$F4\ \underline{P}\ F3$	
procedure $A2(F2)$;	**begin** ··· $F2(3.1)$ ··· **end**;	⇒	$A2\ \underline{F}\ F2$,	$F2\ \underline{F}\ D2$	
procedure $A1(F1)$;	**begin** ··· $F1(A2)$ ··· **end**;	⇒	$A1\ \underline{F}\ F1$,	$F1\ \underline{F}\ D1$,	$D1\ \underline{P}\ A2$
$A1(A3)$;		⇒	$F1\ \underline{P}\ A3$		
	Program			Relations	

Fig. 5. Graph of relations in Figure 4.

$$\begin{tikzcd}
& A1 \arrow[d, "\underline{F}"'] & & & & \\
& F1 \arrow[rr, dashed, "\underline{P}"] \arrow[d] & & A3 \arrow[d] & A4 \arrow[d, "\underline{F}"'] & \\
A2 \arrow[d, "\underline{F}"'] & D1 \arrow[l, dashed, "\underline{P}"'] & & F3 \arrow[ll, dashed, "\text{first iteration}"'] & F4 \arrow[l, dashed, "\underline{P}"'] \arrow[d] & \\
F2 \arrow[rrrr, dashed, "\text{second iteration}"] & & & & D4 \arrow[r, dashed, "\underline{P}"] & SIN
\end{tikzcd}$$

Fig. 6.

P	Actual	Formal	Dummy
Actual	0	0	0
Formal	$P4$	$P5$	$P6$
Dummy	$P7$	$P8$	$P9$

F	Actual	Formal	Dummy
Actual	0	$F2$	0
Formal	0	0	$F6$
Dummy	0	0	0

any call-by-name parameter is definitely a procedure, and the "thunk" is an actual procedure. Since Algol 60 procedure parameters have incomplete parameter specifications (call-by-value, call-by-name is not known for formal procedures), all parameters are potentially call-by-name, the valuing is done inside the procedure. This means all formal parameters are considered formal procedures and any actual parameter compiled as a "thunk" is an actual procedure.

Each of the foregoing relations may be implemented as a bit matrix. If the matrices are partitioned by actual, formal, and dummy procedures, then most of $\underline{F}$ and some of $\underline{P}$ have empty partitions ($\underline{F}$ and $\underline{P}$ are conveniently sparse). This can be used to optimize the evaluation of P, saving space and time. The partitions are numbered left to right, top down, as shown in Figure 6. Step 3 of the algorithm then becomes:

3. $\underline{P6} \leftarrow \underline{F2}'\underline{P4}'\underline{F6} \vee \underline{P6}$
   $\underline{P9} \leftarrow \underline{F6}'\underline{P5}'\underline{F6} \vee \underline{P9}$

Dummy procedures need not be included in the $\underline{C}$ relation so that the inclusion of $\underline{P}$ in C becomes:

$\underline{C4} \leftarrow \underline{C4} \vee \underline{P4}$
$\underline{C5} \leftarrow \underline{C5} \vee \underline{P5}$

Relative Nonrecursion

If a language permits nested procedure declarations, better optimization results can be obtained by modifying the construction of the $\underline{C}$ relation. In Figure 1, the storage for procedure Q can be allocated at the same time P's storage is allocated. This is possible because a call on P precedes every call on Q. We then say that Q is "relatively nonrecursive." In the previous construction of $\underline{C}$, both P and Q would be recursive.

Notice that a call on a procedure directly from the procedure in which it was declared cannot require recursion. This call need not be included in $\underline{C}$. If the called procedure is not declared in the calling procedure, then every procedure in the nest of current procedures out to but not including the procedure in which the called procedure was declared must be placed in the $\underline{C}$ relation to the called procedure. This is because each of these procedures in the nest must be called to perform the considered call and the procedure being called could possibly cause any of them to be called without them first being called. In Figure 7 only procedures $X3$ and $X4$ are recursive.

Fig. 7.

```
procedure X1;
begin
  procedure X2;
  begin
    procedure X3;
    begin
      X4;
    end
    procedure X4;
    begin
      procedure X5;
      begin
        X3;
      end;
      X5;
    end;
    X4;
end;
X2;
end
```

The information needed to construct the modified $\underline{C}$ relation is usually available in a compiler. There is a stack of procedure declarations whose bodies have been entered, but not existed. When a call is encountered, every procedure on the stack down to but not including the procedure in which the called procedure was declared is put in the $\underline{C}$ relation to the called procedure. After recursion analysis is completed, a nonrecursive procedure's storage is allocated with its enclosing recursive procedure, or with the main program if there is no enclosing recursive procedure.

Received May 1975; revised December 1975