



University of  
Arizona

CSC 520

Principles of Programming Languages

Christian Collberg

April 16, 2003

Procedure Calls III

Copyright © 2003 C. Collberg

## Inline Expansion...

- 1. continued...
  - (e) make the jump to the called routine,
  - (f) set up an activation record,
  - (g) execute the body of the called routine,
  - (h) return back to the callee, possibly returning a result,
  - (i) deallocate the activation record.
- 2. Many of these actions don't have to be performed if we inline the callee in the caller, and hence much of the overhead associated with procedure calls is optimized away.
- 3. More importantly, programs written in modern imperative and OO-languages tend to be so littered with procedure/method calls. ...

## Inline Expansion...

- 3. ... This is the result of programming with abstract data types. Hence, there is often very little opportunity for optimization. However, when inlining is performed on a sequence of procedure calls, the code from the bodies of several procedures is combined, opening up a larger scope for optimization.
- There are problems, of course. Obviously, in most cases the size of the procedure call code will be less than the size of the callee's body's code. So, the size of the program will increase as calls are expanded.

## Inline Expansion

- The most important and popular inter-procedural optimization is *inline expansion*, that is replacing the call of a procedure with the procedure's body.
- Why would you want to perform inlining? There are several reasons:
  1. There are a number of things that happen when a procedure call is made:
    - (a) evaluate the arguments of the call,
    - (b) push the arguments onto the stack or move them to argument transfer registers,
    - (c) save registers that contain live values and that might be trashed by the called routine,
    - (d) make the jump to the called routine,

### Inline Expansion...

- Some compilers don't rely on the user to decide on what should be inlined. Instead they use
  1. A static heuristic, such as "procedures which are
    - (a) shorter than 10 lines and have fewer than 5 parameters or
    - (b) are leaf routines (i.e. don't make any calls themselves) are candidates for inlining".
  2. A heuristic based on profiling. After running the program through a profiler we know how many times each procedure is likely to be called from each call site. Only inline small, frequently called procedures.

### Inline Expansion...

- A larger executable takes longer to load from secondary storage and may affect the paging behavior of the machine. More importantly, a routine (or an inner loop of a routine) that fits within the instruction-cache before expansion, might be too large for the cache after expansion.
- Also, larger procedures need more registers (the **register pressure** is higher) than small ones. If, after expansion, a procedure is so large (or contains such complicated expressions) that the number of registers provided by the architecture is not enough, then spill code will have to be inserted when we run out of registers.

### Inline Expansion...

- How do we inline across module boundaries? We need access to the code of the called procedure. If the procedure is declared in a separately compiled module, this code is **not** available. What do we do? Good question...
- What's the difference between inlining and macro expansion? Inlining is performed after semantic analysis, macro expansion before.

### Inline Expansion...

- Several questions remain. Which procedures should be inlined? Some languages (C++, Ada, Modula-3) allow the user to specify (through a keyword or pragma) the procedures that should be eligible for expansions. However, this implies that a given procedure should always be expanded, regardless of the environment in which it is called. This may not be the best thing to do. For example, we might consider inlining a call to  $P$  inside a tightly nested inner loop, but choose not to do so in a module initialization code that is only executed once.

Slide 32-6

Slide 32-4

Slide 32-7

Slide 32-5

## Algorithm...

- $G$  is actually a multigraph since a procedure might make multiple calls to the same procedure.
- Beware of indirect calls through procedure parameters or variables, as well as method invocations!

## Inline Expansion...

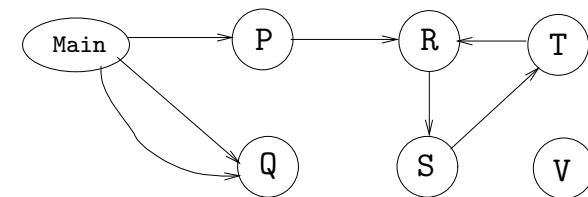
- At which level do we perform the inlining?
  - intermediate code** Most common.
  - source code** Some **source-to-source translators** perform inlining.
  - assembly code** Doable (with some compiler cooperation), but unusual.

## Algorithm...

2. Pick routines to inline. Possible heuristics:
  - (a) Discard recursive routines (Perform a topological sort of the call graph. Cycles indicate recursion.) or just inline them one or two levels deep.
  - (b) Select routines with indegree=1.
  - (c) Select calls to small routines in inner loops.
  - (d) Rely on user-defined **INLINE** pragmas.
  - (e) Use profiling information.
  - (f) Consider effects on caching, paging, register pressure, total code size, ...
  - (g) ...

## Algorithm

1. Build the call graph:
  - (a) Create an empty directed graph  $G$ .
  - (b) Add a node for each routine and for the main program.
  - (c) If procedure  $P$  calls procedure  $Q$  then insert a directed edge  $P \rightarrow Q$ .



## Topological Order...

- Performing the inlining in reverse topological order saves time. For example, expanding V in S before expanding S in R and P is faster than expanding S in P, then S in R, and then V in P and R.
- Note: there is no path `main`  $\rightarrow$  T. Maybe T could be deleted?

## Inlining Example (Original)

```
TYPE T = ARRAY [1..100] OF CHAR;

PROCEDURE P (n : INTEGER; z : T; VAR y :INTEGER);
VAR i : INTEGER;
BEGIN
  IF n < 100 THEN
    FOR i := 1 TO n DO y := z[i] + y; z[i] := 0; ENDFOR
  ENDIF
END P;

VAR S : INTEGER; A : T;
BEGIN P(10, A, S); END
```

## Algorithm...

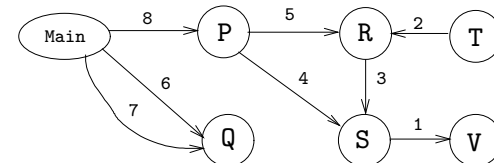
3. **FOR** each call  $P(a_1, \dots, a_n)$  in  $Q$  to inline procedure  $P(f_1, \dots, f_n)$ , in reverse topological order of the call graph **DO**
  - (a) Replace the call  $P(a_1, \dots, a_n)$  with  $P$ 's body.
  - (b) Replace references to call-by-reference formal  $f_k$  with a reference to the corresponding actual parameter  $a_k$ .
  - (c) For each call-by-value formal parameter  $f_k$  create a new local  $c_k$ . Insert code to copy the call-by-value actual  $a_k$  into  $c_k$ . Replace references to the call-by-value formal  $f_k$  with a reference to its copy  $c_k$ .
  - (d) For each of  $P$ 's local variables  $l_k$  create a new local  $v_k$  in  $Q$ . Replace references to local variable  $l_k$  with a reference to  $v_k$ .

## Topological Order

Example:

```
main(){ Q(); ... Q(); };
P(){ R(); ... S(); };
T(){ R();};      R(){S();};
S(){ V();};      Q(){};      V(){};
```

Topological Order:



## Inlining Methods

- Consider a method invocation  $m.P()$ . The actual procedure called will depend on the run-time type of  $m$ .
- If more than one method can be invoked at a particular call site, we have to inline all possible methods. The appropriate code is selected code by branching on the type of  $m$ .
- To improve on method inlining we would like to find out when a call  $m.P()$  can call exactly one method.

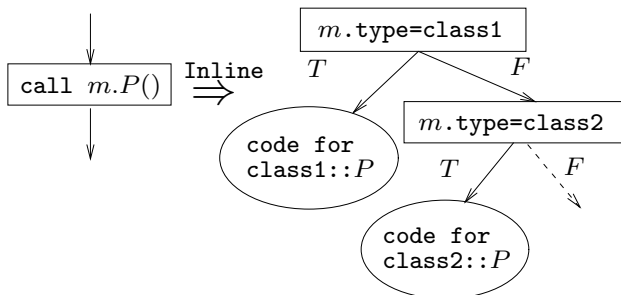
## Inlining Example (Expanded)

```
TYPE T = ARRAY [1..100] OF CHAR;
VAR   S, $n, $i   : INTEGER;
      A, $z       : T;

BEGIN
  $n := 10;
  copy($z, A, 100);

  IF $n < 100 THEN
    FOR $i := 1 TO $n DO
      S := $z[$i] + S; $z[$i] := 0; ENDFOR
    ENDIF
  END
```

## Inlining Methods...



## Inlining Example (Optimized)

```
TYPE T = ARRAY [1..100] OF CHAR;

VAR   S, $i       : INTEGER;
      A, $z       : T;

BEGIN
  copy($z, A, 100);

  FOR $i := 1 TO 10 DO
    S := $z[$i] + S;
    $z[$i] := 0;
  ENDFOR
END
```

## Type Hierarchy Analysis...

```
TYPE T = CLASS [] [  
    METHOD M (); BEGIN END M; ];  
TYPE S = CLASS EXTENDS T [] [  
    METHOD N (); BEGIN END N;  
    METHOD M (); BEGIN END M; ];  
VAR x : T; y : S;  
BEGIN  
    x.M();  $\Leftarrow S_{T,M} = \{T.M, S.M\}$   
    y.M();  $\Leftarrow S_{S,M} = \{S.M\}$   
END;
```

## Inlining Methods — Example

```
TYPE T = CLASS [f : T] [  
    METHOD M (); BEGIN END M;  
    ];  
TYPE S = CLASS EXTENDS T [  
    ] [  
    METHOD N (); BEGIN END N;  
    METHOD M (); BEGIN END M;  
    ];  
VAR x : T; y : S;  
BEGIN  
    x.M();  
    y.M();  
END;
```

## C Preprocessor

- The C preprocessor (`cpp`) is a program that preprocesses a C source file before it is given to the C compiler.
- The preprocessor's job is to remove comments and apply preprocessor directives that modify the source code.
- Preprocessor directives begin with `#`, such as the directive `#include <stdio.h>` in `hello.c`.

## Type Hierarchy Analysis

- For each type  $T$  and method  $M$  in  $T$ , find the set  $S_{T,M}$  of method overrides of  $M$  in the inheritance hierarchy tree rooted in  $T$ .
- If  $x$  is of type  $T$ ,  $S_{T,M}$  contains the methods that can be called by  $x.M()$ .
- We can improve on type hierarchy analysis by using a variant of the Reaching Definitions data flow analysis.

## C Preprocessor Examples...

```
#define ARRAY_SIZE 1000
#ifdef DEBUG
    printf("got here!")
#endif

#if defined(DEBUG)
    #define Debug(x) printf(x)
#else
    #define Debug(x)
#endif

Debug(("got here!"));
```

## Macros vs. Inlined Procedures

- A macro is expanded **before** syntactic and semantic analysis takes place.
- An inline function is expanded **after** semantic analysis.
- Hence, the body of a macro is analyzed statically in the environment of the caller, the body of an inlined procedure is analyzed in the environment of the callee (itself).

## C Preprocessor...

Some popular ones are:

**#include <file>** The preprocessor searches the system directories (e.g. /usr/include) for a file named **file** and replaces this line with its contents.

**#define word rest-of-line** Replaces word with rest-of-line throughout the rest of the source file.

**#if expression ... #else ... #endif** If expression is non-zero, the lines up to the **#else** are included, otherwise the lines between the **#else** and the **#endif** are included.

## C Preprocessor Examples

```
#define ARRAY_SIZE 1000
char str[ARRAY_SIZE];

#define MAX(x,y) ((x) > (y) ? (x) : (y))
int max_num;
max_num = MAX(i,j);

#define DEBUG 1
```

Slide 32-28

## Macros vs. Inlined Procedures...

- If foo is an inline procedure it will print "yes".
- If foo is a macro it will not compile.

```
var x : boolean := true; [inline|macro] foo();  
  if x then print "yes" else print "no"; end foo;  
  
begin  
  var x : integer := 5;  
  foo(); end.
```

Slide 32-29

## Readings and References

- Read Scott: pp. 441-442, 301-303.
- Read the Dragon book: 530-532, 585-602.