CSc 553

Principles of Compilation

29 : Optimization IV

Department of Computer Science
University of Arizona

collberg@gmail.com

Introduction

Inline Expansion

Inline Expansion I

- 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:
     1. evaluate the arguments of the call,
     2. push the arguments onto the stack or move them to argument transfer registers,
     3. save registers that contain live values and that might be trashed by the called routine,
     4. make the jump to the called routine,

Inline Expansion II

- 1. continued. . .
     5. make the jump to the called routine,
     6. set up an activation record,
     7. execute the body of the called routine,
     8. return back to the callee, possibly returning a result,
     9. 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 III

- 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 IV

- 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 V

- 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.

Inline Expansion VI

- 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
     1. shorter than 10 lines and have fewer than 5 parameters or
     2. 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 VII

- 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.
- At which level do we perform the inlining?

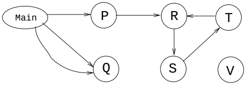
  intermediate code Most common.

  source code Some **source-to-source translators** perform inlining.

  assembly code Doable (with some compiler cooperation), but unusual.

Algorithm I

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



- 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!

Algorithm II

2. Pick routines to inline. Possible heuristics:
   1. Discard recursive routines (Perform a topological sort of the call graph. Cycles indicate recursion.) or just inline them one or two levels deep.
   2. Select routines with indegree=1.
   3. Select calls to small routines in inner loops.
   4. Rely on user-defined **INLINE** pragmas.
   5. Use profiling information.
   6. Consider effects on caching, paging, register pressure, total code size, ...
   7. ...

Algorithm III

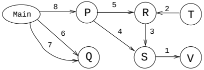
3. **FOR** each call $P(a_1, \cdots, a_n)$ in Q to inline procedure $P(f_1, \cdots, f_n)$, in reverse topological order of the call graph **DO**
    1. Replace the call $P(a_1, \cdots, a_n)$ with P's body.
    2. Replace references to call-by-reference formal f_k with a reference to the corresponding actual parameter a_k.
    3. 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.
    4. 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:



- Performing the inlining in reverse topological order saves time: 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` → `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
```

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
```

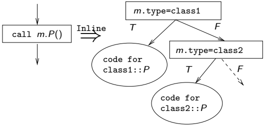
Inlining Example (Optimized)

```
        ⇓ Optimize

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
```

Inlining in OO Languages

Inlining Methods I

- 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*.



Inlining Methods II

- To improve on method inlining we would like to find out when a call m.P() can call exactly one method.

```
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;
```

Inlining Methods III

Inheritance 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 $x.M()$.

Example

```
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();  ⇐ $S_{T,M} = \{T.M, S.M\}$
   y.M();  ⇐ $S_{S,M} = \{S.M\}$
END;
```

Inlining Methods IV — Intraprocedural Type Propagation

- We can improve on type hierarchy analysis by using a variant of the Reaching Definitions data flow analysis.

```
TYPE T = CLASS [][METHOD M (); ···];
TYPE S = CLASS EXTENDS T [][
         METHOD M (); BEGIN END M;];
VAR p :  S; o :  T;
BEGIN
   p := NEW S;
   IF e THEN
      o := NEW T; o.M();
   ELSE
      o := p; o.M();
   ENDIF;
   o.M();
   o := NARROW(o, S); o.M();
END;
```

Inlining Methods V

$$\begin{aligned} \texttt{out}[B] &= \texttt{Gen}[B] \cup (\texttt{in}[B] - \texttt{Kill}[B]) \\ \texttt{in}[B] &= \bigcup_{\texttt{preds } P \texttt{ of } B} \texttt{out}[P] \end{aligned}$$

- We'll use sets of pairs $\langle t, S\rangle$, where S is the set of possible types of variable t.
- The equations are defined over statements, not basic blocks.
- The function `TypeOf(x)` is the set of current types of `x`.
- $\boxed{\texttt{Gen}[B] = \{\langle v, \{S\}\rangle\}}$ statement B contains an operation that guarantees that v will have type S.
- $\boxed{\texttt{Out}[B] = \{\langle v, \{S\}\rangle, \langle u, \{S, T\}\rangle\}}$ after statement B, v will have type S and u will have type S or T.

Inlining Methods VI

$\boxed{\texttt{Gen}(v{:=}\ \texttt{NEW}\ t) = \langle v, \{t\}\rangle}$ After v `:= NEW` t, v must have the type t.

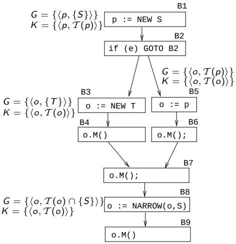
$\boxed{\texttt{Gen}(v{:=}u) = \langle v, \texttt{TypeOf}(u)\rangle}$ After an assignment v `:=` u, the set of possible types of v is the same as the current set of possible types of u.

$\boxed{\texttt{Gen}(v{:=}\ \texttt{NARROW}(u, T)) = \langle v, \texttt{TypeOf}(u) \cap T\rangle}$ After an assignment $v{:=}\ \texttt{NARROW}(u, T)$, the set of possible types of v contains those current possible types of u that are also a subtype of T.

$\boxed{\texttt{Kill}(v{:=}u) = \langle v, \texttt{TypeOf}(v)\rangle}$ After an assignment $v{:=}u$, v may not have the same type that it had before.

$\boxed{\texttt{Kill}(v{:=}\ \texttt{NEW}\ t) = \langle v, \texttt{TypeOf}(v)\rangle}$

$\boxed{\texttt{Kill}(v{:=}\ \texttt{NARROW}(u, T)) = \langle v, \texttt{TypeOf}(v)\rangle}$



$G = \{\langle p, \{S\}\rangle\}$
$K = \{\langle p, \mathcal{T}(p)\rangle\}$

B1 `p := NEW S`

B2 `if (e) GOTO B2`

$G = \{\langle o, \mathcal{T}(p)\rangle\}$
$K = \{\langle o, \mathcal{T}(o)\rangle\}$

$G = \{\langle o, \{T\}\rangle\}$
$K = \{\langle o, \mathcal{T}(o)\rangle\}$

B3 `o := NEW T`

B5 `o := p`

B4 `o.M()`

B6 `o.M();`

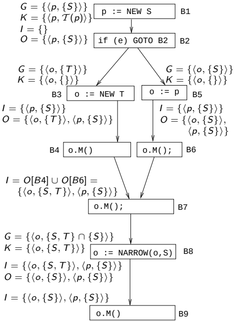
B7 `o.M();`

$G = \{\langle o, \mathcal{T}(o) \cap \{S\}\rangle\}$
$K = \{\langle o, \mathcal{T}(o)\rangle\}$

B8 `o := NARROW(o,S)`

B9 `o.M()`

NOTE: $\mathcal{T}(x) = \texttt{TypeOf}(x)$



$G = \{\langle p, \{S\}\rangle\}$
$K = \{\langle p, \mathcal{T}(p)\rangle\}$
$I = \{\}$
$O = \{\langle p, \{S\}\rangle\}$

B1 `p := NEW S`

B2 `if (e) GOTO B2`

$G = \{\langle o, \{T\}\rangle\}$
$K = \{\langle o, \{\}\rangle\}$

$G = \{\langle o, \{S\}\rangle\}$
$K = \{\langle o, \{\}\rangle\}$

B3 `o := NEW T`

B5 `o := p`

$I = \{\langle p, \{S\}\rangle\}$
$O = \{\langle o, \{T\}\rangle, \langle p, \{S\}\rangle\}$

$I = \{\langle p, \{S\}\rangle\}$
$O = \{\langle o, \{S\}\rangle,$
$\langle p, \{S\}\rangle\}$

B4 `o.M()`

B6 `o.M();`

$I = O[B4] \cup O[B6] =$
$\{\langle o, \{S, T\}\rangle, \langle p, \{S\}\rangle\}$

B7 `o.M();`

$G = \{\langle o, \{S, T\} \cap \{S\}\rangle\}$
$K = \{\langle o, \{S, T\}\rangle\}$

B8 `o := NARROW(o,S)`

$I = \{\langle o, \{S, T\}\rangle, \langle p, \{S\}\rangle\}$
$O = \{\langle o, \{S\}\rangle, \langle p, \{S\}\rangle\}$

$I = \{\langle o, \{S\}\rangle, \langle p, \{S\}\rangle\}$

B9 `o.M()`

Summary

Readings and References

- Read the Tiger book, Section 15.4. This describes inlining in functional languages, but the ideas are the same.