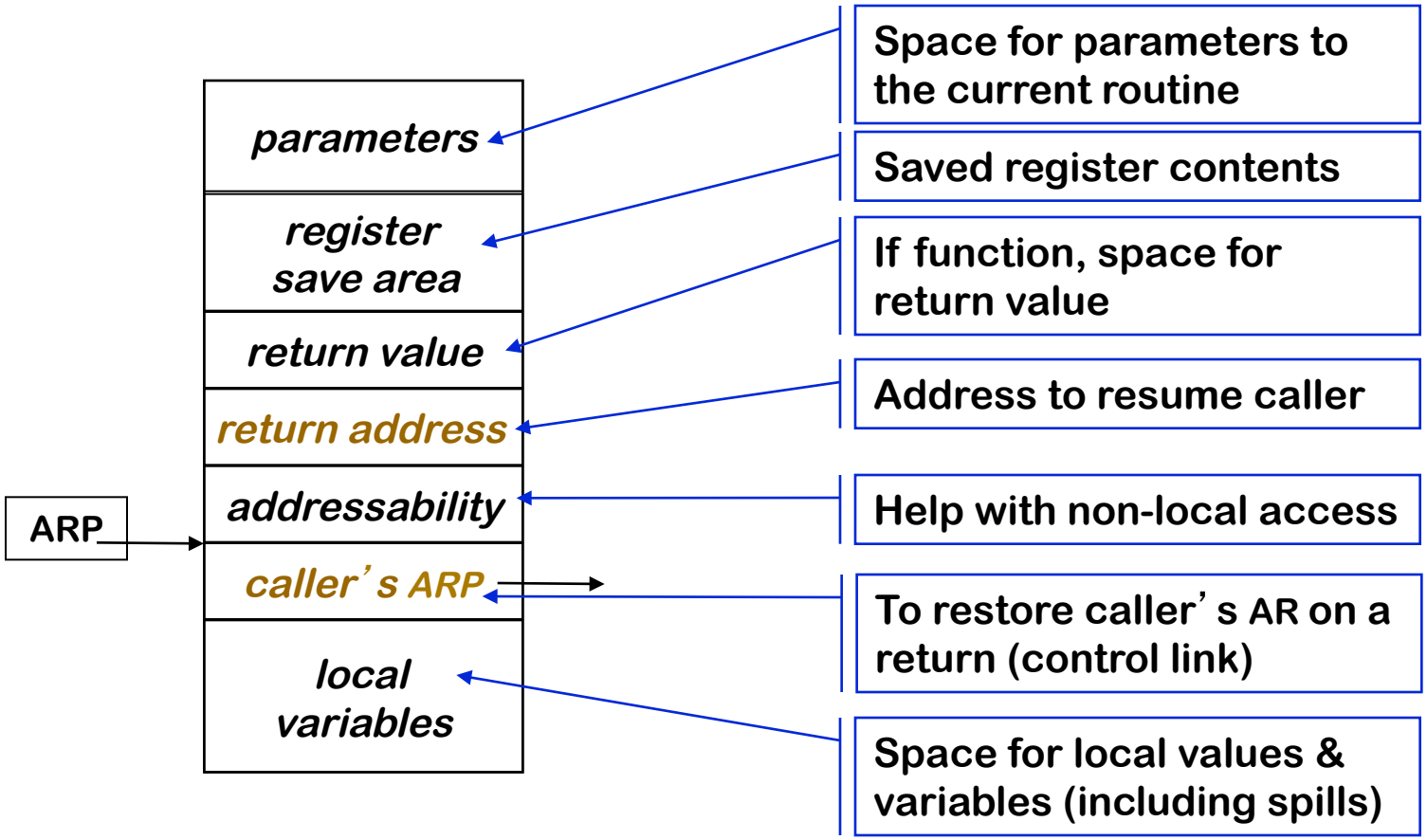
RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

CS415 Compilers

Interprocedure Analysis and CSE Optimization

These slides are based on slides copyrighted by Keith Cooper, Ken Kennedy & Linda Torczon at Rice University

Review - Activation Record Basics



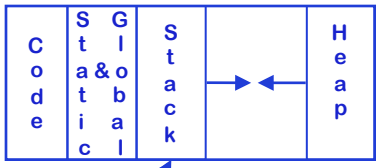
One AR for each invocation of a procedure

Activation Record Details

Where do activation records live?

- If lifetime of AR matches lifetime of invocation, *AND*
- If code normally executes a "return"

⇒ Keep ARs on a stack



- If a procedure can outlive its caller, *OR*
- If it can return an object that can reference its execution state

⇒ ARs must be kept in the heap

- If a procedure makes no calls

⇒ AR can be allocated statically

Efficiency prefers static, stack, then heap

Activation Record in the Heap

- If a procedure can outlive its caller, *OR*
- If it can return an object that can reference its execution state

⇒ ARs must be kept in the heap

Consider functional languages (ML, Scheme):

```
fun f(x) =
  let
    fun g(y) = x * y
  in
    g
  end
```

The code:

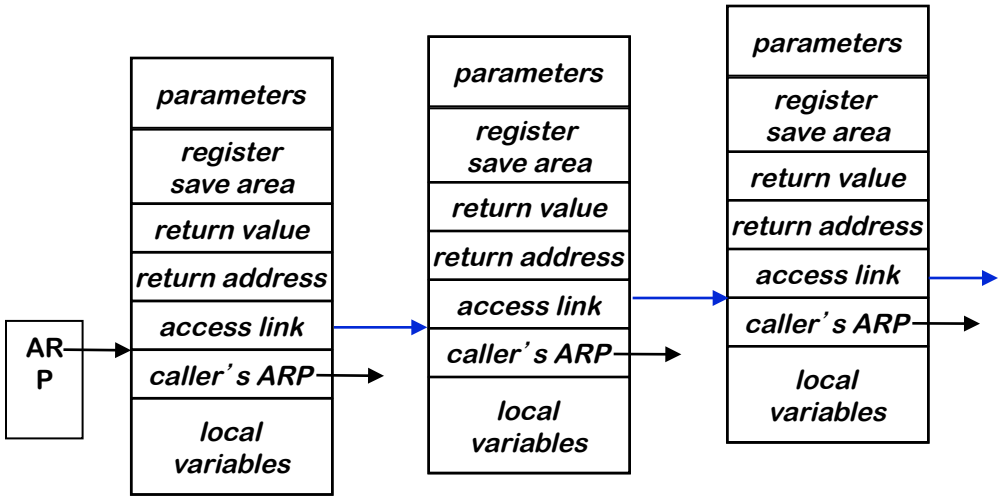
```
val z = f(4)
val w = z(5)
```

Establishing Addressability

Using access links (static links)

- Each AR has a pointer to most recent AR of immediate lexical ancestor (mylevel - 1)
- Lexical ancestor need not be the caller

Some setup cost on each call



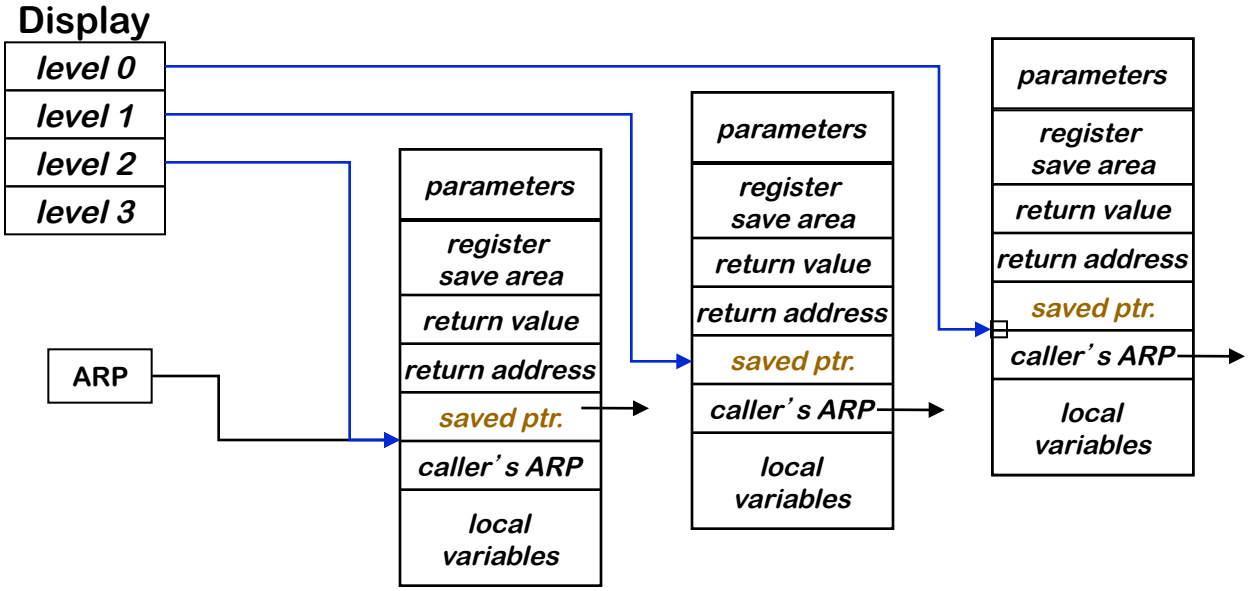
- Reference to <*p*,16> runs up access link chain to *p*
- Cost of access is proportional to lexical distance

Establishing Addressability

Using a display

- Global array of pointer to nameable ARs
- Needed ARP is an array access away

Some setup cost on each call



- Reference to <p,16> looks up p's ARP in display & adds 16
- Cost of access is constant (ARP + offset)

Procedure Linkages

How do procedure calls actually work?

- At compile time, callee may not be available for inspection
  - → Different calls may be in different compilation units
  - → Compiler may not know system code from user code
  - → All calls must use the same protocol

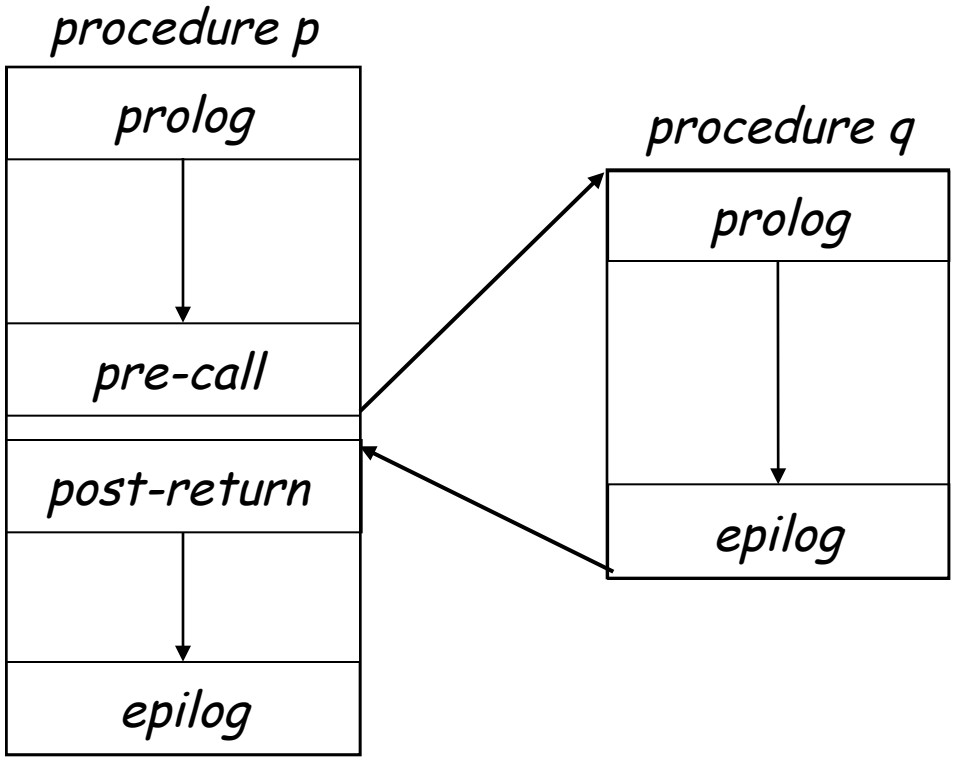
Compiler must use a standard sequence of operations

- Enforces control & data abstractions
- Divides responsibility between caller & callee

Usually a system-wide agreement *(for interoperability)*

Procedure Linkages

Standard procedure linkage

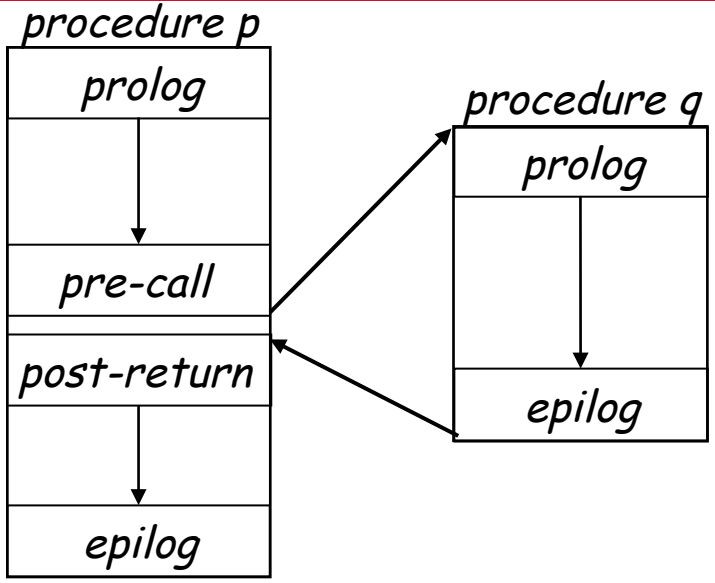


Procedure has
- standard prolog
- standard epilog

Each call involves a
- pre-call sequence
- post-return sequence

These are completely predictable from the call site ⇒ depend on the number & type of the actual parameters

Procedure Linkages

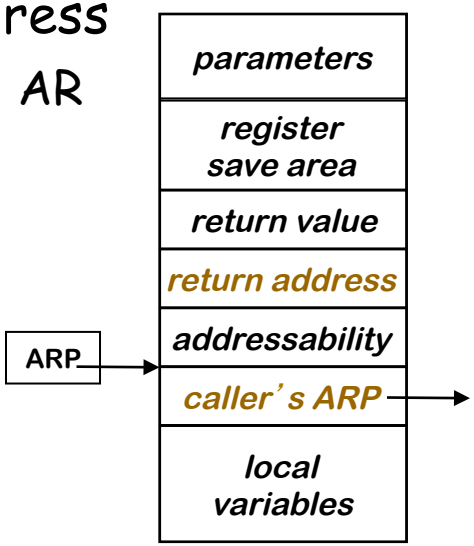


Pre-call Sequence

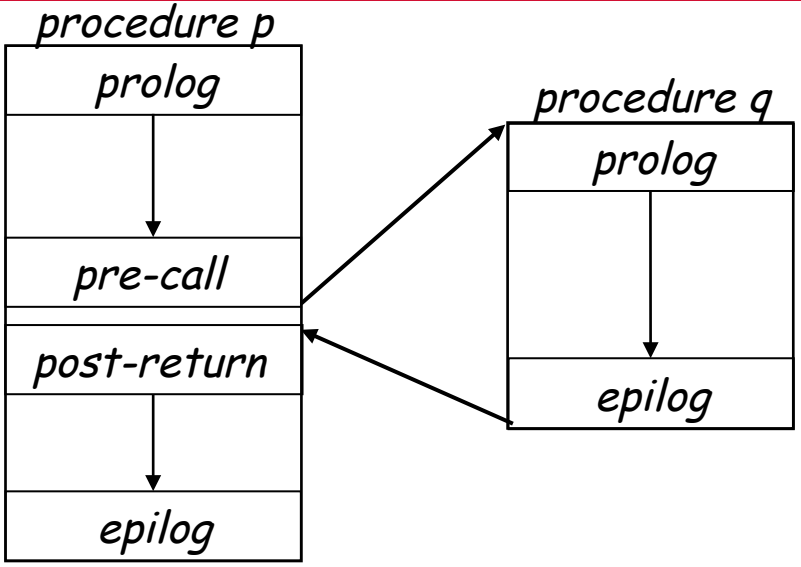
- Sets up callee's basic AR
- Helps preserve its own environment

The Details

- Allocate space for the callee's AR
  - → except space for local variables
- Evaluates each parameter & stores value or address
- Saves return address, caller's ARP into callee's AR
- Save any caller-save registers
  - → Save into space in caller's AR
- Jump to address of callee's prolog code



Procedure Linkages

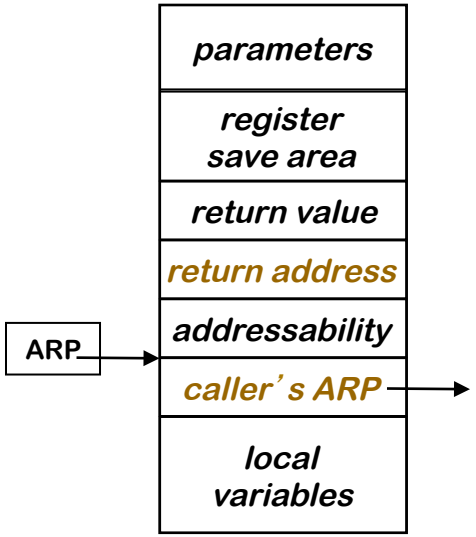


Post-return Sequence

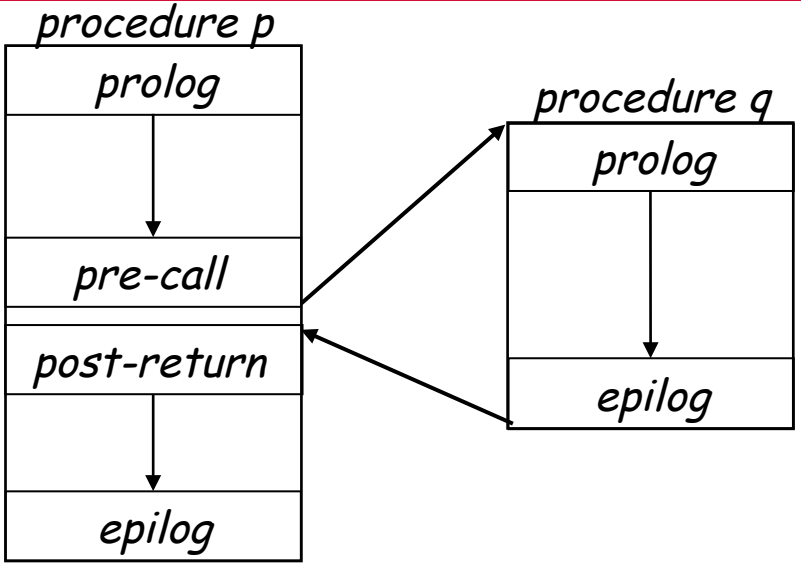
- Finish restoring caller's environment
- Place any value back where it belongs

The Details

- Copy return value from callee's AR, if necessary
- Free the callee's AR
- Restore any caller-save registers
- Copy back call-by-value-result parameters
- Continue execution after the call



Procedure Linkages

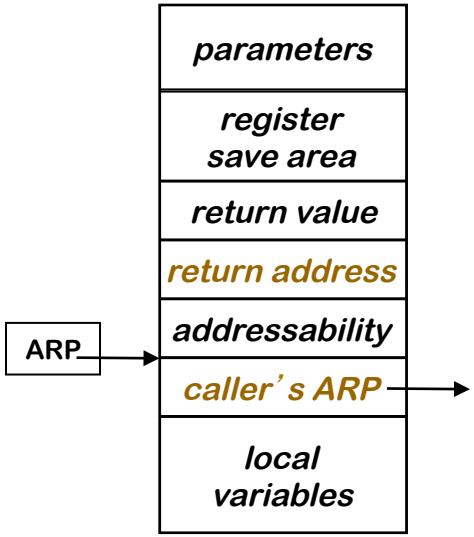


Prolog Code

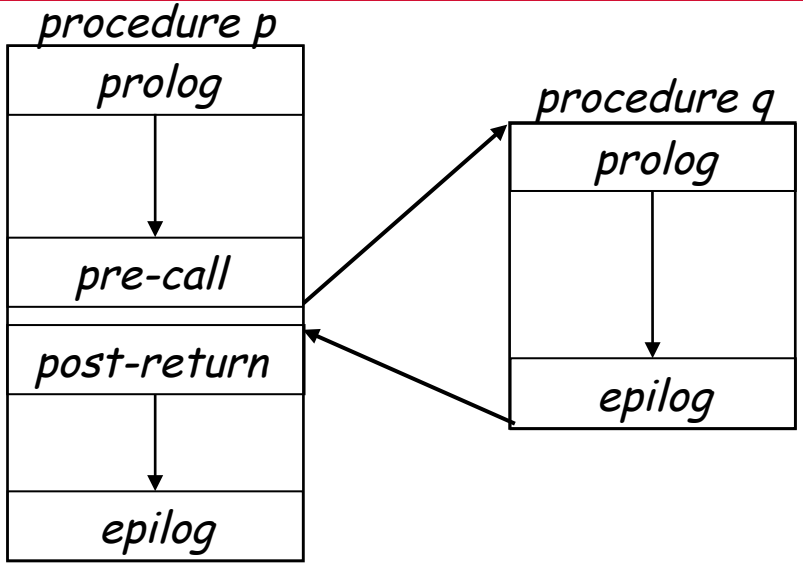
- Finish setting up callee's environment
- Preserve parts of caller's environment that will be disturbed

The Details

- Preserve any callee-save registers
- Allocate space for local data
  - → Easiest scenario is to extend the AR
- Handle any local variable initializations



Procedure Linkages

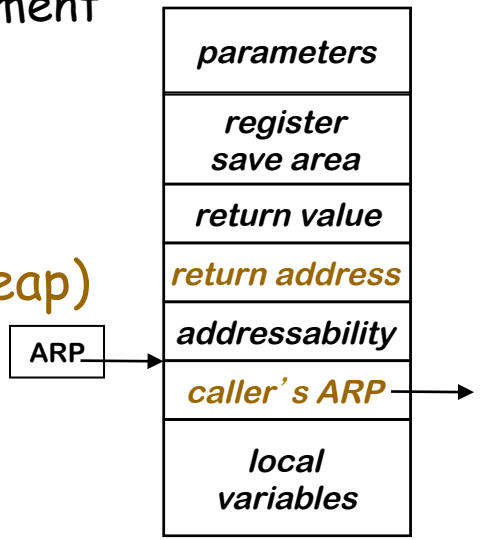


Epilog Code

- Wind up the business of the callee
- Start restoring the caller's environment

The Details

- Store return value?
  - → Some implementations do this on the return statement
  - → Others have return assign it & epilog store it into caller's AR
- Restore callee-save registers
- Free space for local data, if necessary (on the heap)
- Load return address from AR
- Restore caller's ARP
- Jump to the return address



A Coin Has Two Sides

Compiler uses a **standard** sequence of operations

- Enforces control & data abstractions
- Divided responsibility between caller & callee
- Caller assumes no knowledge of callee, vice versa

Separation of concerns v.s. Whole program optimization

- Benefits of whole program analysis
  - → Can produce better code around call sites (saves and stores)
  - → Can provide shaper global analysis
  - → Can present the optimizer with more context
  - → Can provide tailored copies of procedures

What can be so difficult?

```
procedure joe(i,j,k)
        l <- 2 * k
        if (j == 100)
            then m <- 10 * j
            else m <- i
        call ralph(l,m,k)
        o <- m * 2
        q <- 2
        call ralph(o,q,k)
        write q, m, o, l
```

```
procedure main
    call joe( 10, 100, 1000)

procedure ralph(a,b,c)
    b <- a * c / 2000
```

Will always print out "1000, 1000, 2000, 2000". Can skip the function call operations at runtime if well optimized.

What happens at a procedure call?

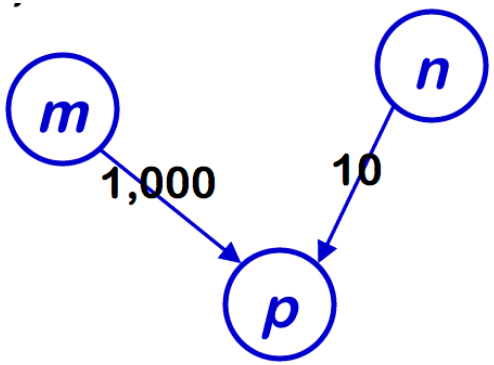
- Use worst case assumptions about side effects
- Leads to imprecise intra-procedural information
- Leads to explosion in intra procedural def-use chains

What can be so difficult?

Different calls to p have different properties

- Frequency of the call
- Environment that p inherits
  - → Mapping from parameters to values (& locations)
  - → Constant values & known values
  - → Size of task
- Surrounding execution context
  - → Is call in a parallel loop?
  - → Which registers are unused?
- Procedure-valued parameters

May want to optimize distinct calls separately!



***p* must function correctly in *both* contexts**

Interprocedure Analysis

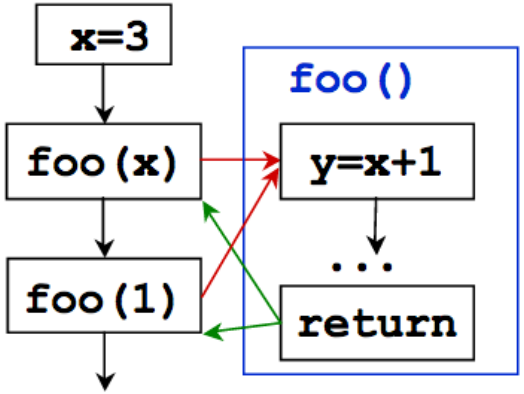
Types of Interprocedure Analysis

- Flow-sensitive v.s. flow-insensitive
  - → Flow-sensitive: analysis at every program point
    Need iterative data-flow analysis (IDFA)
  - → Flow-insensitive: computer answer for an entire procedure
    Might be less accurate than flow-sensitive analysis
- Context-sensitive v.s. context-insensitive
  - → Context-sensitive: assumes knowing context of each caller
  - → Context-insensitive: analysis independent of callers
- Path-sensitive v.s. path-insensitive
  - → Path-sensitive: Computes one answer for every execution path
  - → Path-insensitive: merge information from all paths

Supergraph Construction

Compose CFGs for the entire program

- Connect call nodes to entry of callees
- Connect return nodes of callees back to calls
- Control-flow supergraph



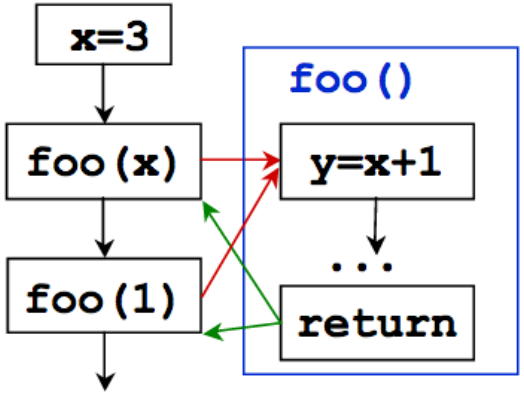
Supergraph Construction

Compose CFGs for the entire program

- Connect call nodes to entry of callees
- Connect return nodes of callees back to calls
- Control-flow supergraph

Pros

- Simple
- Direct use of intraprocedure analysis



Cons

- Loss of accuracy
- Performance
- Requires no separate compilation

Full Context-sensitive Analysis

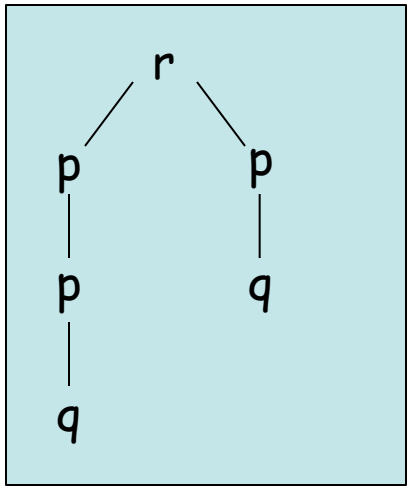
```
int  r (…) { // declaration
     int   d, s;

     int q (x,y) // declaration
           int x,y;
     {
        return x + y + d;
     }

     int p (a,b,c) // declaration
           int a, b, c;
     {
       int   d;
       if (…)
          d = q (c,b); // call
       else
          d = p (a, d, c); // call
     }
     s = p(10, d, s); // call
     s = p(11, s, d); // call
}
```

Invocation graph

- Distinguish all call-chains
- Analyze callee for all distinct call paths
- Precise but expensive



Procedure Summaries

Compute summary information for each procedure

- Summarize effect of called procedure for callers
- Summarize effect of callers for called procedure
- Tracks information flow into or out of a procedure

- Examples
  - → MAY-ALIAS: the set of formals that may be aliased to globals and each other, i.e., f(x,x)
  - → MUST-ALIAS, CONST
  - → Side-effect summaries
    - MOD: the set of variables possibly modified (def) by a call
    - REF: the set of variables possibly read (use) by a call
    - KILL: the set of variables killed by a procedure (liveness)

Alternatives to Interprocedure Analysis

- Inlining
  - → Replace a call with procedure body
  - → Pros: reduce call overhead, exposed call context & side effects
  - → Cons: code bloat, library source not necessarily available, not for recursion!

- Procedure cloning/specialization
  - → Create a customized version of procedure for particular call sites
  - → Middle grounds between inlining and interprocedure optimization
  - → Pros: less code bloating
  - → Cons: interprocedure analysis still necessary

Optimizations

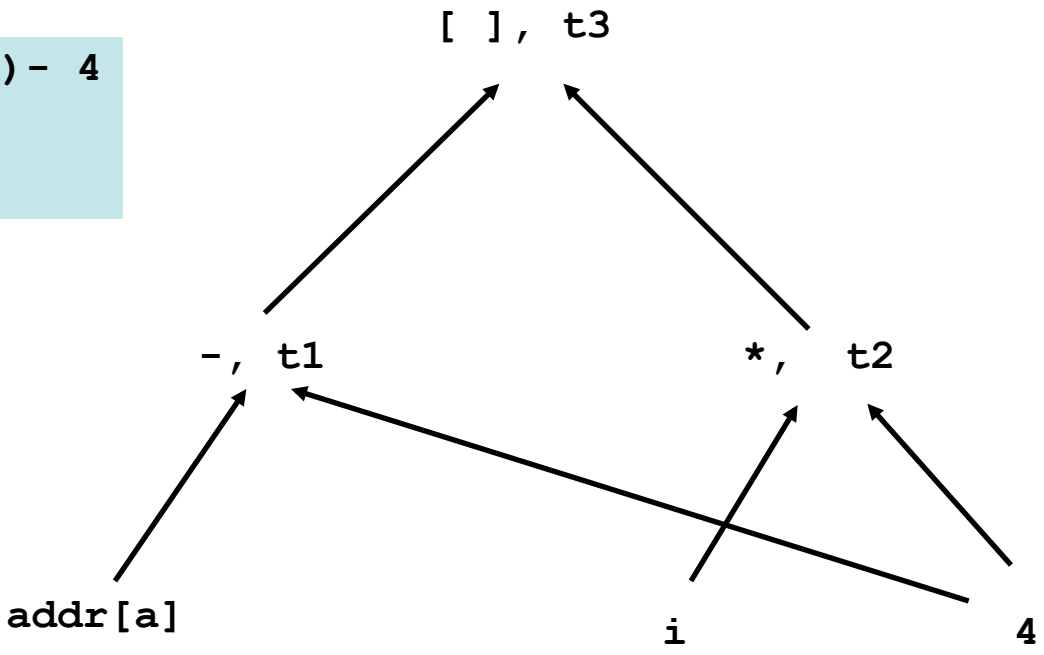
Basic Block DAG Construction

Optimization: Local Common Subexpression Elimination (CSE)

Source code: **a(i) * a(i)**

a(i)

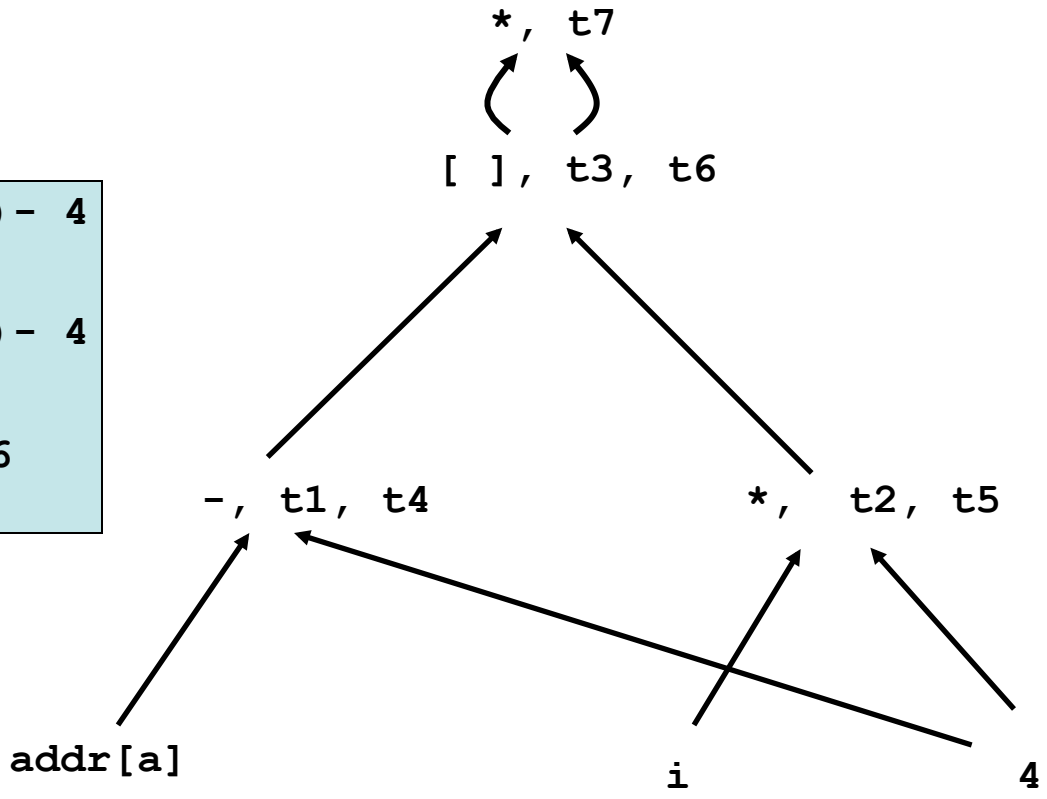
```
4.      t1 = addr(a)- 4
5.      t2 = i * 4
6.      t3 = t1[t2]
    . . .
```



Basic Block DAG Construction

a(i) * a(i)

```
4.      t1 = addr(a)- 4
5.      t2 = i * 4
6.      t3 = t1[t2]
7.      t4 = addr(a)- 4
8.      t5 = i * 4
9.      t6 = t4[t5]
10.     t7 = t3 * t6
    . . .
```

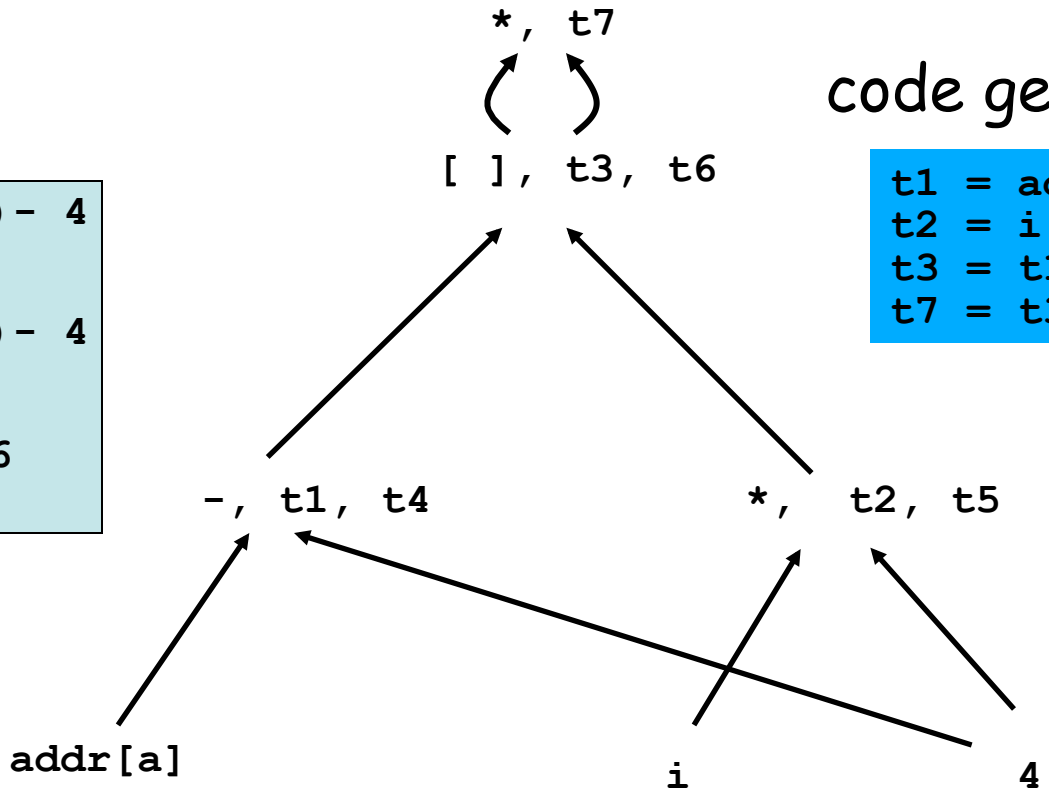


Basic Block DAG Construction

Local common subexpression elimination (CSE):

a(i) * a(i)

```
4.      t1 = addr(a)- 4
5.      t2 = i * 4
6.      t3 = t1[t2]
7.      t4 = addr(a)- 4
8.      t5 = i * 4
9.      t6 = t4[t5]
10.     t7 = t3 * t6
    . . .
```



code generated:

```
t1 = addr[a]-4
t2 = i * 4
t3 = t1[t2]
t7 = t3 * t3
```

Basic Block DAG Construction

How to add a subexpression into a partially constructed DAG:

A = B + C

Is there a node already for B + C?

- If so, add A to its list of labels.
- If not:
  - is there a node labeled B already?

    If not, create a leaf labeled B.
  - Is there a node labeled C already?

    If not, create a leaf labeled C.
  - Create a node labeled A, for +, with left child B and right child C.

How to do this? HASHING <op, node(opd1), node(opd2)>

Basic Block DAG Construction

DAG Construction Algorithm

How to add a subexpression into a partially constructed DAG:

A = B + C

Is there a node already for B + C? <+,node(B),node(C)> defined?

- If so, add A to its list of labels.
- If not:
  - is there a node labeled B already? node(B) defined?

    If not, create a leaf labeled B.
  - Is there a node labeled C already? node(C) defined?

    If not, create a leaf labeled C.
  - Create a node labeled A, for +, with left child B and right child C. Create node(+) with children node(B), node(C)

How to do this? HASHING <op, node(opd1), node(opd2)>