Interprocedural Analysis

Course so far:

- Control Flow Representation
- Dataflow Representation
  - SSA form
  - Classic DefUse and UseDef Chains
- Optimizations
- Scheduling
- Register Allocation
- Just-In-Time Compilation
- GC
- Alias analysis

All of these have been for an individual procedure - *intraprocedural* analysis.

Today:

Interprocedural analysis (across/between procedures)

- Brief overview

Dealing with Procedures

Terminology

```
int a, e                  // globals
procedure foo(var b, c) // formal args
    b := c
end
program main
    int d                 // locals
    foo(a, d)             // call site with
end                       //   actual args
```

In procedure body

- formals and/or globals may be *aliased* (two names refer to same location)
- formals may have constant value

At procedure call

- global vars may be modified or used
- actual args may be modified or used

Conservative assumptions for procedures may significantly inhibit optimization

Dealing with Procedures

Aliasing example

```
int a
foo(a, a)
...
procedure foo(var b, c)
a := 1      // what is b, c?
b := 2      // what is a, c?
c := 3      // what is a, b?
d := a + b  // what is d?
end
```

Side effect example

```
a := 1
foo(a)      // what is used/killed?
b := a      // what is b?
```

Constants example

```
foo(1)
...
procedure foo(a)
if (a = 1)   // what is a's value?
   ...
```


Characterizing an Analysis

Definiteness

- May
  - captures possibility
  - optimizations "must account for"
- Must
  - captures certainty
  - optimizations "may rely on"

Flow sensitivity

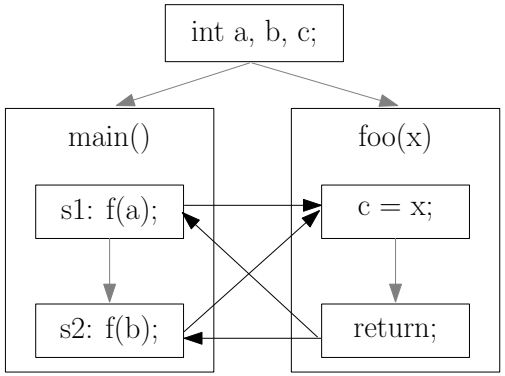
- Sensitive (consider control flow)
  - Determine value for each program point
  - Requires data-flow analysis
  - Precise
- Insensitive (ignore control flow)
  - Determine value for whole procedure
  - Can compute in linear time
  - Imprecise

Characterizing an Analysis

Context sensitivity

- Sensitive (aka *polyvariant analysis*)
  - Determine values for each calling context
  - Must re-analyze callee for each caller
- Insensitive (aka *monovariant analysis*)
  - Summarize values for all calling contexts
  - Cheap(ish) but imprecise

Recall: Procedure body and procedure call



Why don't many compilers use IP analysis?

- Inlining - poor woman's interprocedural analysis.
  Is inlining an alternative, or is it part of IP analysis and transformation?
- Separate compilation with IP analysis requires *recompilation analysis*.
- IP analysis can be expensive.
  e.g., Flow-sensitive kill is intractable.
- Benefits of IP analysis on optimization have not been well explored, especially for scalar machines.

  Constants: Grove/Torczon'93, Metzer/Stoud'93
  Dependence analysis: Paek et al.'98
  Li et al.'90, Havlak/Kennedy'90
  Parallelizing Compilers: Hall et al.'95, McKinley'93
- It is hard – solutions can be complex.
  Especially true for flow-sensitive problems.

Interprocedural Compilation

Goals

- enable standard optimizations even in presence of procedure calls
- perform additional optimizations not possible for single procedures
- reduce call overhead for procedures

Issues

- compilation speed
- re-engineering compiler
- separate compilation
- recompilation analysis
- libraries (no source code)
- dynamic shared objects (DSOs)

Driving applications

- parallelization
- data layout optimizations

Interprocedural Compilation

Interprocedural analyses (IPA)

- MAY-ALIAS — may be aliased
- MUST-ALIAS — must be aliased
- MOD — may be modified
- REF — may be referenced
- USE — may be used before kill
- KILL — must be killed
- AVAIL — must be available
- CONSTANT — must be constant

Interprocedural optimizations

- inlining
- cloning
- register allocation
- loop transformations

The Call Graph

$G = (V, E, start)$ where each procedure is a unique vertex V and each call site is represented by an edge between the *caller* or invoking procedure and the *called* procedure.

- (u, v) = call from u to v
- cycles represent recursion
- the graph is less likely to be reducible than the *cfg*

Example:

```
procedure r
    integer i,j,k
    ...
    call s(i,j)
    ...
    call s(i,k)
    procedure s(a,b)
        integer a, b
        swap (a, b)
        ...
    end procedure s
end procedure r
```


Parameter Binding

At procedure boundaries, we need to translate formal arguments of procedure to actual arguments of procedure at call site. We call this *parameter binding*.

Example

```
int a,b
program main           // MOD(foo) = b
   foo(b)              // REF(foo) = a,b
end
procedure foo (var c) // GMOD(foo)= b
   int d               // GREF(foo)= a,b
   d := b
   bar(b)              // MOD(bar) = b
end                    // REF(bar) = a
procedure bar (var d)
   if (...)            // GMOD(bar)= d
      d := a           // GREF(bar)= a
end
```

GMOD, GREF = side effect of procedure

MOD, REF = effect at call site (perhaps specific to the call)

Direction of Interprocedural Analysis

Bottom-up (MOD, REF, USE, KILL)

- summarizes effect of call

```
procedure foo
   int a
   bar(a)
end
```

Top-down (MAY-ALIAS)

- summarizes information from caller

```
int a,b
procedure foo(var int b, c)
   ...
end
foo(a,a)
foo(a,b)
```

Bi-directional (AVAIL, CONSTANT)

- information to/from caller and callee

```
int a,b,c,d
procedure foo(e)
   a := b + c
   d := e
end
foo(1)
```

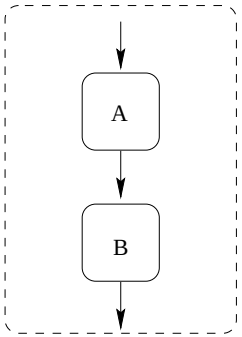

Precision of Interprocedural Analysis

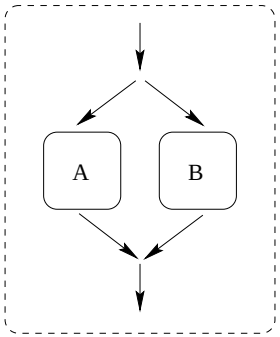
Flow-insensitive (MAY-ALIAS, MOD, REF)

- result not affected by control flow in procedure

Flow-sensitive (USE, KILL, AVAIL)

- result affected by control flow in procedure





Info	B(A( ))	A( ) ∧ B( )
MOD		
REF		
USE		
KILL		

Flow-insensitive Aliases without Pointers

For languages like Fortran, aliases arise only due to parameter passing.

$\Longrightarrow$ We can compute solutions efficiently by separating local and global portions of solution

1. calculate where aliases are introduced
   (needs only local information)
2. propagate aliases

Reduce complexity further by finding global-formal and formal-formal aliases separately

Flow-insensitive alias algorithm

1. find global-formal aliases
2. find formal-formal aliases
3. combine

K. Cooper and K. Kennedy, "Fast Interprocedural Alias Analysis," Sixteenth ACM Symposium on the Principles of Programming Languages (POPL), January 1989.

Alias Analysis

Variables x, y are *aliased* in procedure p if they may refer to the same memory location in p
$\Longrightarrow$ alias pair $\langle x,y \rangle \in$ MAY-ALIAS

Aliases are introduced when

- the same variable (formal, local, or global) is passed to two different locations in the same call
  $\Longrightarrow$ the formals become an alias pair

```
procedure bar(int c)
   foo(c,c)
end
procedure foo(var a, b)
   ...              // <a,b> in ALIAS
end
```

- there is a visible (global) variable in both the caller and callee passed as an actual parameter
  $\Longrightarrow$ global and formal become an alias pair

```
int b
foo(b)
procedure foo(var a)
   ...              // <a,b> in ALIAS
end
```

Alias Analysis

Aliases are propagated when p calls q and

- two formals in p which are aliases are used as actuals in the same call q,

  $\Longrightarrow$ the formals in q become aliases

```
p(a,a)
procedure p(var b, c)
   q(b,c)
end
procedure q(var d, e)
   ...               // <d,e> in ALIAS
end
```

- when a variable aliased to a global is passed as a parameter to q,

  $\Longrightarrow$ formal in q becomes aliased to global

```
int a
p(a)
procedure p(var b)
   q(b)
end
procedure q(var c)
   ...             // <a,c> in ALIAS
end
```


Alias Analysis (cont)

Aliases are propagated when p calls q and

- when two actuals are aliased to the same global
  $\Longrightarrow$ the formals become aliases

```
int a
p(a)
procedure p(var b)
   q(a,b)
end
procedure q(var c, d)
   ...              // <c,d> in ALIAS
end
```

Finding Global-Formal Aliases

Improve efficiency of propagating aliases by using the *binding graph* (β)

Binding graph

1. nodes are formal parameters and globals
2. $\exists$ edge $\langle x \rangle$ to $\langle y \rangle$ iff some call binds x to y
3. use local information to mark where aliases to globals are introduced

Alias algorithm (part 1)

- collapse strongly connected components (SCC)
  (same aliases hold for all parameters in SCC)
- maintain set of aliases for each formal
- propagate aliases to all reachable nodes in β
- results in all global-formal aliases

Finding Formal-Formal Aliases

Improve efficiency of propagating aliases by using the *pair binding graph* (π)

Pair binding graph

- nodes are pairs of formal parameters
- $\exists$ edge $\langle x,y \rangle$ to $\langle z,w \rangle$ iff some call
  binds x to z and y to w

Alias algorithm (part 2)

1. initialize local information when aliases are introduced
   (add edges when two actuals are aliased to same global)
2. propagate aliases to all reachable nodes in π
3. results in all formal-formal aliases

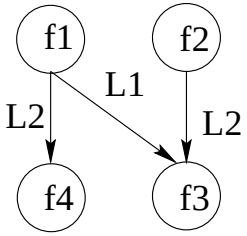
Alias algorithm (part 3)

1. combine global-formal and formal-formal aliases to yield all aliases

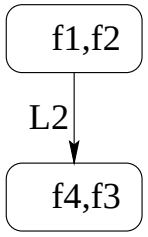
Flow-insensitive Alias Example

```
     int x, y, z
     procedure p(var f1, f2)
L1:     q(f1, z)
L2:     q(f2, f1)
     end
     procedure q(var f3, f4)
        ...
     end
L3:  p(x, z)
L4:  q(y, y)
```

	f1	f2	f3	f4
ALIAS				



Binding Graph



Pair Binding Graph

Interprocedural Constant Propagation

Properties

- find formals with constant values
- ignores interprocedural control flow
- uses interprocedural aliases, side effects
- uses intraprocedural control flow
- bidirectional analysis on call graph

Procedure summaries

- avoid revisiting procedure bodies
- $pcp(y,s)$

  returns constant value of actual parameter y at call site $s=(p,q)$ if it can be determined from local analysis in p with interprocedural MOD
- $rpcp(x,p)$

  returns constant value of formal parameter x if it can be determined from local analysis in p with interprocedural MOD
- used to build *jump functions*

Jump Functions

- J_s^y: *forward jump function*

  for actual parameter y at call site $s = (p,q)$, J_s^y gives the value of y in p

  – *support*(J_s^y) is the exact set of p's formals whose values on entry to p are used in the computation of J_s^y

- R_p^x: *backward jump function*

  for each formal parameter x modified by p, R_p^x returns the value of x on return from p

  – *support*(R_r^x) is the exact set of formals of p whose values on entry to p are used to compute R_p^x

- different levels of precision

D. Callahan, K. Cooper, K. Kennedy, and L. Torczon, "Interprocedural constant propagation," ACM SIGPLAN Symposium on Compiler Construction, June 1986

Jump Functions

1) Literal constant jump function

$$J_s^y = \begin{cases} c & \text{if } y \text{ is literal constant } c \text{ at } s \\ \bot & \text{otherwise} \end{cases}$$

Example

```
procedure foo(int x)
  bar(1)
end
procedure bar(int y)
```

2) Intraprocedural constant jump function
(including interprocedural ALIAS, MOD)

$$J_s^y = \begin{cases} c & \text{if } pcp(y,s) = c \\ \bot & \text{otherwise} \end{cases}$$

Example

```
procedure foo(int x)
  x = 2
  bar(x)
end
procedure bar(int y)
```

Jump Functions

3) Pass-through parameter jump function

$$J_s^y = \begin{cases} c & \text{if } pcp(y,s) = c \\ x & \text{if } y = x \text{ at } s = (p,q) \text{ and } x \text{ is formal of } p \\ \bot & \text{otherwise} \end{cases}$$

Example

```
procedure foo(int x)
  bar(x)
end
procedure bar(int y)
```

4) Polynomial parameter jump function

$$J_s^y = \begin{cases} c & \text{if } pcp(y,s) = c \\ f(support(J_s^y)) & \text{if } y \text{ can be represented as a polynomial } f \text{ of } support(J_s^y) \\ \bot & \text{otherwise} \end{cases}$$

Example

```
procedure foo(int x)
  x = (x*x) + (2*x) + 1
  bar(x)
end
procedure bar(int y)
```


Jump Functions

5) Reverse jump function

$$R_s^y = \begin{cases} c & \text{if } rpcp(x,p) = c \\ g(support(R_s^y)) & \text{if } x \text{ can be represented as a polynomial } g \text{ of } support(R_s^y) \\ \bot & \text{otherwise} \end{cases}$$

Example

```
procedure foo(var x, y)
  x = y + 1
end
```

Complexity of jump functions

- literal constant, interprocedural constant, pass-through parameter jump functions

$$O(\sum_s \sum_y cost(J_s^y)) \text{ time}$$

- polynomial parameter, reverse jump functions

$$O(\sum_s \sum_y cost(J_s^y) * \text{support}(J_s^y)) \text{ time}$$

Next Time

Read: Goff, Tseng, & Kennedy, Practical Dependence Testing, ACM Conference on Programming Language Design and Implementation, Toronto, Canada, June 1991.