Interprocedural Dataflow Analysis
Propagating information across procedure boundaries is useful.

- Optimize caller using information about callee
  
  ```
  x := 2
  CALL f(x) //call by reference
  ...x...  //is x equal to 2 here?? Need to look at f
  ```

- Optimize callee using information about callers
  
  ```
  ... PROCEDURE f(a,b)
  CALL f(x,5) .......
  ...
  ```

- We might generate specialized code for f in which b is 5.
- We might *clone* f and specialize clone.
- We might inline f.

Where do such opportunities arise? (i) calling library code (ii) object-oriented programs.
Facts about interprocedural dataflow analysis

- Significantly harder asymptotically and to implement than intraprocedural dataflow analysis
  - Intraprocedural analysis: unknowns are lattice values.
  - Interprocedural analysis: unknowns are functions on lattice values.
- Aliasing: different program names for same location
- Strategies for dealing with complexity:
  - Invent special-purpose algorithms that work for important special cases (e.g., if the lattice is finite or of bounded height)
  - Use general-purpose techniques that compute approximate but conservative solutions
Game plan for inter-procedural analysis lectures:

- Start with call-by-value language
  
  Key problem: solving dataflow function equations

- Call-by-value + global variables

  Some problems have structure we can exploit to speed up analysis.

- Call-by-reference language

  Additional problem: aliasing
Let us begin with simple program model in which there is no aliasing

```plaintext
MAIN()
    var p,q;
    p = read();
    q = f(p,3);
    ...
PROCEDURE f(x,y)
    var z,a;
    if (x > 0) z = x;
    a = y;
    ...f(y,z)...
```

- no higher-order procedures
- no data structures
- no global variables
- call-by-value
- assignments in procedure modify locals and parameters
- recursion is allowed

Check: no aliasing
Key data structure: Call (multi)graph

- **Structure:**
  Nodes: one for each procedure
  Edges: from node f to node g if procedure f may invoke procedure g

```
MAIN()
  ... f(...)
  .... g(...)
  ...f(...)
  ...
  f(..)
  ...f(...)
  ...g(...)
  g(..)
  ...h(...)
  h(..)
  ...g(...)

Program               Call Graph
```
• **Algorithm:** Building call graph is trivial if there are no higher-order procedures.

• **Use:** Call graph plays a role sort-of like that of control flow graph in intraprocedural analysis, but not quite....
Context-insensitive analysis

One obvious approach: reduce interproc case to intraproc case

forward dataflow problem:
- merge information from all call sites of procedure at START
- copy dataflow information coming out of END to all return sites

PROCEDURE f (..)
..... id(2) ....
.....id(3).....

PROCEDURE id(n)
return n;

Problem: information propagates along impossible interprocedural paths
such as blue edge into id and red edge out of id

This loses precision: in our example, we would not detect that id(2) is 2!

However, this reduction of inter-procedural analysis to the intra-procedural case
is safe.

It is called context-insensitive analysis.
Context-sensitive analysis

Do not mix dataflow information from different call-sites of a procedure

One implementation of context-sensitive analysis:
- model each program procedure by a function on dataflow values
- to analyze dataflow effect of a call to procedure g, dataflow analyzer invokes the associated function, passing it some dataflow values and getting dataflow values back
- solves problem of avoiding dataflow mixing

```
PROCEDURE f (..)
    .... id(2) ....
    ......id(3).....

PROCEDURE id(n)
    return n;
```

```
function fc(..)
    ...idc(2)....
    ...idc(3)....

function idc(n)
    return n;
```

Dataflow functions for constant propagation

Another implementation: build the composite graph as in context-insensitive analysis but propagate dataflow values together with a "tag" that identifies call sequence that generated that value (Sharir and Pnueli)
Main problem with context-sensitive analysis: termination

- Intuitively, we are doing something similar to a symbolic execution of program.
- Analysis must terminate even if program execution does not terminate!
- Difficulty with recursive procedures: analysis usually requires symbolic execution of both sides of conditionals, so how do we ensure termination? Here’s an example where program terminates but analysis would not.
procedure main() {
    var p;
    p = read();
    return f(3,p);
}

procedure f(n,p) {
    if (n > p)
        then return f(n-1,p);
    else return 1;
}

function mainc() {
    p = bottom;
    p = T;
    return fc(3,p);
}

function fc(n,p) {
    t1 = t2 = t3 = bottom;
    if (n > p)
        then t1 = fc(n-1,p);
    else t2 = 1
    t3 = join(t1,t2);
    return t3;
Source of termination problem:

The recursive definition of $f_c$ in previous slide is really an equational definition of $f_c$: interpreting this definition as an “executable function” gets us into trouble in general.

Two important special cases:

- **No recursion in program**: no problem with non-termination in interpreting equational definitions as functions. Determining there is no recursion: call-graph should have no cycles. This idea does not work even in the presence of *static* recursion which even FORTRAN allows.*Static recursion*: text of program has recursive calls; *Dynamic recursion*: at runtime, two or more activations of a procedure coexist at some point in time.
- **Domain is finite**: solve equations iteratively by tabulating values of functions.
Solving recursive function equations by tabulation

- necessary condition: finite lattice

- Example: constant propagation in which all values other than 0 and 1 are set to $T$.

```
function fc(n,p)
    {t1 = t2 = t3 = bottom;
     if (n>p) then t1 = fc(n-1,p);
        else t2 = 1;
     t3 = join(t1,t2);
    return t3;}
```

Compute a sequence of approximations to fc as follows:

```
fc[0] (n,p) = ⊥

fc[i+1](n,p) =
    {t1 = t2 = t3 = bottom;
     if (n>p) then t1 = fc[i](n-1,p);
        else t2 = 1;
     t3 = join(t1,t2);
    return t3;}
```

For our example:

```
fc(n,p) = 1
```

Computing each element of sequence: make a table of output for each possible input value
Checking convergence: same table is obtained for two successive elements in sequence.
Termination: from monotonicity and finiteness of domain.
What do we do if we have recursion and domain is not finite?

Usual strategy: Replace recursive dataflow function with non-recursive conservative approximation

1. Identify recursive calls in call graph, and approximate their return values to $T$. Then, solve resulting acyclic problem.

```plaintext
function fc(n,p){
    t1 = t2 = t3 = bottom;
    if (n > p)
        then t1 = fc(n-1,p);  --->
        else t2 = 1;
    t3 = join(t1,t2);
    return t3;}
```

Recursive call identification: back edges in DFS of call-graph (not necessarily unique)
2. Solve context-insensitive problem and use result to approximate the effect of recursive calls.

- Value on "return edge" will be an upper bound of possible return values
- Replace recursive call with that value and solve acyclic context-sensitive problem

```
fc(n,p) {
    t1 = t2 = t3 = bottom;
    if (n > p)
        then t1 = fc(n-1,p);
        else t2 = 1;
    t3 = join(t1,t2);
    return t3;
}
```
Adding global variables to program model:

```plaintext
GLOBAL G1, G2, G3
MAIN()
    var p,q
    p = f(G1,3); //S1 => MOD-S1 is {p,G2}
    ...
PROCEDURE f(x,y)
    var z,a
    if (x > 0) G2 := x; // S2 => MOD-S2 is {G2}
    z = G1; // S3 => MOD-S3 is {z}
    ...f(G1,z)...
```

Still no aliasing.

**Inter-procedural dataflow analysis: simple extension of call-by-value case.**

(eg) Context-sensitive inter-procedural constant propagation: dataflow function for procedure will have one additional parameter for each global variable.
Exploiting structure in inter-procedural dataflow analysis

- Just as in intra-procedural case, inter-procedural problems may have structure that can be exploited to speed up solution.
- Exploiting intra-procedural structure: as before
- Inter-procedural structure: in the call graph
- For many problems, we can exploit strongly connected components in call graph to speed up analysis (eg. MOD computation)
Interprocedural dataflow problem: computing \textsc{Mod} [Banning]

For any statement $s$, the set \textsc{Mod}-$s$ is the set of variables visible to $s$ that \textit{may} be modified directly or indirectly by execution of $s$.

GLOBAL $G_1$, $G_2$, $G_3$
MAIN()
    var $p,q$
    $p = f(G_1,3); // S_1 \Rightarrow \text{MOD-S}_1$ is $\{p,G_2\}$
...
PROCEDURE $f(x,y)$
    var $z,a$
    if ($x > 0$) $G_2 := x; // S_2 \Rightarrow \text{MOD-S}_2$ is $\{G_2\}$
    $z = G_1; // S_3 \Rightarrow \text{MOD-S}_3$ is $\{z\}$
    ...$f(G_1,z)\ldots$

Auxiliary sets: For any procedure $f$, \textsc{GMod-$f$} is the set of global variables that may be modified directly or indirectly by invoking $f$.

In example, \textsc{GMod-MAIN} $=$ $\{G_2\}$, and \textsc{GMod-$f$} $=$ $\{G_2\}$

For any statement $s$, \textsc{Mod-$s$} is the union of

- set of variables that may be modified directly in statement (\textsc{IMod})
- set of globals that may be modified directly or indirectly by procedure invocations in $s$ (\textsc{GMod})

So given \textsc{GMod} sets, \textsc{Mod} sets are easy to compute.
How do we compute GMOD sets?

Write down a set of lattice equations and solve them.

- **Lattice**: power-set of global variables
- **Equations**: if procedure f has assignments to globals $G_i, G_j, \ldots$ and it may invoke procedures $g, h, \ldots$ equation for GMOD-f is

$$GMOD-f = \{G_i, G_j, \ldots\} \cup GMOD-g \cup GMOD-h, \ldots$$
GLOBAL G1, G2, G3;
MAIN ()
  ... f(...)
  .... g(...)
  ...f(...)
  ...
  f(..)
  G1 := ...
  ...f(...)...
  ...g(...)...
  g(..)
  ...h(...).
  G1 := ...
  h(..)
  G3 := ...
  ...g(...)...

Program
Observations

- We can use any iterative technique we discussed in intra-procedural case to solve these inter-procedural equations.
- Is there structure that can be exploited to reduce number of iterations? Yes!!
  - In our problem, information flows from invoked procedure to invokee.
  - So consider equations in “reverse invocation order”. See next slide.
GLOBAL G1, G2, G3;
MAIN ()
  ... f(…)
  ... g(…)
  ...f(…)
  ...
  f(…)
  G1 := ...
  ...f(…)
  ...g(…)
  g(…)
  ...h(…).
  G1 := ...
  h(…)
  G3 := ...
  ...g(…)...

[Call Graph]

Reverse invocation order

GMOD-main = GMOD-f U GMOD-g
GMOD-f = {G1} U GMOD-g U GMOD-f
GMOD-g = {G1} U GMOD-h
GMOD-h = {G3} U GMOD-g

Program
**Further simplification**: note that GMOD sets for all procedures in a single scc of call graph must be identical.

So collapse equations for all procedures in a single scc into a single equation!

\[
\text{GMOD-}g = \{G1\} \cup \text{GMOD-}h \implies \text{GMOD-}gh = \{G1,G3\} \cup \text{GMOD-}gh \\
\text{GMOD-}h = \{G3\} \cup \text{GMOD-}g
\]

Solving single equation for least solution: trivial! Just drop the recursive term.

So in example, \(\text{GMOD-}g = \text{GMOD-}h = \{G1,G3\}\)
Summary: GMOD/MOD computation for call-by-value language

- Write down GMOD equations for the program.
- Partition equations by scc’s in call graph.
- Collapse equations in each scc into a single equation.
- In reverse topological order of acyclic condensate of call graph, read off solutions to GMOD equations.
- For each statement, compute MOD set.

Complexity: $O(\text{size of program} \times \text{number of variables})$

Note: we can exploit scc’s any time we have a set of equations (eg, block triangular systems in linear algebra)
Running example:

GLOBAL G1, G2, G3;
MAIN ()
    ... f(...) 
    .... g(...) 
    ...f(...) 
    ...
    f(..)
    G1 := ...
    ...f(...) 
    ...g(...) 

    g(..)
    ...
    h(...)
    G1 := ...

    h(..)
    G3 := ...
    ...

Program
Call-by-reference

Complications:

1. Effect of a procedure is not just globals it modifies but also what happens to parameters

   \[ x := f(G1, w) \]  //to compute MOD, we need to know what happens to G1 and w!

2. **Aliasing**: two program names for same location

   ```
   GLOBAL G1, G2
   procedure f(x, y)
   x := 3;  //S1
   ....
   procedure g(z, w)
   ....f(G1, G2)....f(z, z)...f(w, z)
   ```

   For first call to f, x and y are not aliases
   For second call to f, x and y are aliases
   For third call to f, x and y are aliases if w and z are aliases!

   What is MOD-S1???

Let us handle these problems one at a time.
Aliasing

- **MUST-ALIAS**: two program names that definitely refer to the same memory location.

```plaintext
GLOBAL G1;
procedure f(x) {
    x = 7;
}
procedure main() {
    f(G1);
    print(G1);
}
```

MUST-ALIAS is an equivalence relation on names.
• **MAY-ALIAS**: two program names that may or may not refer to the same memory location. MAY-ALIASing usually arises from MUST-ALIASes through loss of information such as when we merge information along different program paths.

GLOBAL G1,G2;

procedure f(x) { <---- x and G1 are MAY-ALIAS’s within f
    x = 7;}

procedure main() {
    f(G1) + f(G2);
    print(G1); }

MAY-ALIAS relation is reflexive and symmetric but not necessarily transitive.
Representation of aliasing information: alias pairs

GLOBAL G1,G2;

procedure f(x) { //ALIAS = {<x,x>,<x,G1>,<x,G2>}
    x = 7;
}

procedure main() {
    f(G1) + f(G2);
    print(G1); }

If <a,b> does not occur in an ALIAS relation, variables a and b are definitely not aliased at the point in program where ALIAS relation holds.

Some people also store may/must flag with each alias pair.

Alias pairs are an example of store-free alias representation.

Store-based alias representation: see when we talk about pointer analysis.
Using ALIAS relations in inter-procedural dataflow analysis:

For our language model, all statements in a procedure have same alias relation (not true when we have pointers as in C).

Modify the dataflow transfer functions of statement with alias information

- Constant propagation:

  procedure f(x,y) { function fc(x,y) {
    .... ....
    x := e; V-out = {
      let n = Eval(e,V-in);
      return V-out where
      V-out[i] = V-in[i] if i is not aliased with x
      V-out[i] = join(Vin[i],n) if i is MAY-ALIASed with x
      V-out[i] = n if i is MUST-ALIASed with x
    }
    .... ....
  } }
• MOD computation: close affected variables under aliasing

GLOBAL G1,G2
procedure f(x,y)
  x := 3; // MOD = {x,G1,y}
  ....
procedure g(z,w)
  f(G1,G2)...f(z,z)...f(w,z)
Computing alias relation for our program model: treat as a dataflow problem...

- Compute one alias set for each procedure.
- Each call-site has an associated transfer function that generates output alias set from alias set of caller (see next slide).
- Alias set of procedure = union of alias sets generated at its call sites.
- Transfer functions are monotonic and lattice is finite.
procedure f(x,y) {
    ...
}
procedure g(a,b,c) {  //suppose alias set is A
    ....f(p,q)...  //output alias set is B
}

Transfer(A, list of actuals, list of formals, globals)
B = { };
    for each actual parameter p in call do {
        if (p is a global variable) then append <p,x> to B;
        else for each tuple <p,V> in A do
            if V is a global variable then append <x,V> to B;
    }
    for each pair (p,q) of actual parameters in call do {
        if ((<p,q> is in A) or (p and q are same variable))
            then append <x,y> to B where x/y are formals bound to p/q by call.
        return B;
Example: [adapted from Banning]

GLOBAL x,y,z;

procedure MAIN() ALIAS-MAIN = {}
p1(y); ALIAS-C1 = {<y,y1>}
p1(x); ALIAS-C2 = {<x,y1>}
p3(y,z); ALIAS-C3 = {<y,x3>,<z,y3>}

procedure p3(x3,y3) ALIAS-p3 = ALIAS-C3 U ALIAS-C6
  x3 := ...;
y3 := ...;
x  := ...;

procedure p1(y1) ALIAS-p1 = ALIAS-C1 U ALIAS-C2 U ALIAS-C7 U ALIAS-C8
  p2(z); ALIAS-C4 = {<x2,z>}
p2(y1); ALIAS-C5 = Transfer(ALIAS-p1,(y1)->(x2),{x,y,z})
p3(y1,y); ALIAS-C6 = Transfer(ALIAS-p1,(y1,y)->(x3,y3),{x,y,z})

procedure p2(x2) ALIAS-p2 = ALIAS-C4 U ALIAS-C5
  p1(x2); ALIAS-C7 = Transfer(ALIAS-p2,(x2)->(y1),{x,y,z})
p1(z); ALIAS-C8 = {<z,y1>}

Solution: ALIAS-p1 = {<y,y1>,<x,y1>,<z,y1>}
ALIAS-p2 = {<y,x2>,<x,x2>,<z,x2>}
ALIAS-p3 = {<y,y3>,<x,x3>,<z,x3>,<y3,x3>,<y,x3>,<z,y3>}
CALL p1(y) CALL p1(x) CALL p3(y,z)
CALL p2(z) CALL p2(y1)

MAIN

ALIAS-MAIN = {}

ALIAS-p1

ALIAS-p2

ALIAS-p3
More efficient ways of computing alias sets:

- Determine which globals are aliased with which formals by using **binding graph** (see next slide).
  - Graph has a node for each global and formal parameter; if global/formal v1 is passed to formal v2, put an edge from v1 to v2.
  - All formals reachable from node for global g are aliased to g.

- Determining which formals are aliased to each other: use **pairwise binding graph**. Left to reader.
Structure of binding graph:
- one node for each reference parameter and global
- if procedure f passes its reference parameter/global r1 to procedure g as reference parameter r2, put an edge from r1 to r2

Finding aliases between globals and formals:
all reference parameters reachable from a global variable in the binding graph may be aliased to that global

In our example:
- x may be aliased to y1, x3, x2
- y may be aliased to y1, y3, x2, x3
- z may be aliased to y1, x2, x3

Binding graph
Concern 2: GMOD must tell us what happens to globals AND parameters.

One model: make GMOD into a function from variables to variables

\[ w := f(G4,s); \quad //S1 \]

procedure \( f(x,y) \) => GMOD-\( f(v1,v2) \)

\[
\begin{align*}
\text{var a,b;} & \quad \text{return \{G1,v1\}} \\
G1 & := 5; \\
\text{if (y>x) x:= 7;} \\
\ldots
\end{align*}
\]

Intuition: (assuming \( s,w \) have no non-trivial aliases)

\[
\text{MOD-S1 = \{w\} U GMOD-}f(G4,s) = \{w\} U \{G1,G4\} = \{w,G1,G4\}
\]
Example: [adapted from Banning]

GLOBAL x,y,z;

procedure MAIN() GMOD-MAIN()
    p1(y);
    p1(x);
    p3(y,z);
    return GMOD-p1(y) U GMOD-1(x) U GMOD-p3(y,z)

procedure p3(x3,y3) GMOD-p3(m,n)
    x3 := ...;
    y3 := ...;
    x := ...;
    return {m,n,x}

procedure p1(y1) GMOD-p1(m)
    p2(z);
    p2(y1);
    p3(y1,y);
    return GMOD-p2(z) U GMOD-p2(m) U GMOD-p3(m,y)

procedure p2(x2) GMOD-p2(m)
    p1(x2);
    return GMOD-p1(m) U GMOD-p1(z)

Need to solve recursive functional equations.
Without recursion, we can use “interpret equations as program” trick.

Another approach: since our lattice is finite, we can always use tabular approach.

\begin{verbatim}
GMOD-p3(m,n)
    return {m,n,x}
GMOD-p1(m)
    return GMOD-p2(z) U GMOD-p2(m) U GMOD-p3(m,y)
GMOD-p2(m)
    return GMOD-p1(m) U GMOD-p1(z)
\end{verbatim}

Iterations:

\begin{verbatim}
1 2 3 ... final

GMOD-p3(m,n):  {}  {m,n,x}  {m,n,x}  ...  {m,n,x}
GMOD-p2(m):    {}  {}  {}  ...  {m,y,x,z}
GMOD-p1(m):    {}  {}  {m,y,x}  ...  {m,y,x,z}
\end{verbatim}
Exploiting structure for computing GMOD

• **call-by-value:** we found GMOD sets without iteration
  • find scc’s in call graph
  • for each scc, compute union of side-effects to globals by procedures in scc
  • propagate GMOD sets in reverse invocation order

• can we use this trick for call-by-reference as well?

• **Problem:** GMOD sets contain both globals and parameters, so GMOD sets of mutually recursive procedures will be different in general.

• **Idea:** separate computation of side-effects to parameters from side-effects to globals
Side-effects to parameters:

procedure f(a,b) ...

$\text{RMOD}_f\text{-}a = \text{true}$ if execution of $f$ modifies $a$ directly or indirectly

GLOBAL $x,y,z$;
procedure MAIN()
    p1(y);
    p1(x);
    p3(y,z);
procedure p3(x3,y3) $\text{RMOD}_{p3}\text{-}x3 = \text{true}$
    x3 := ...;
    y3 := ...;
    x := ...;
procedure p1(y1) $\text{RMOD}_{p1}\text{-}y1 = \text{true}$ $\text{RMOD}_{p2}\text{-}x2 \lor \text{RMOD}_{p3}\text{-}x3$
    p2(z);
    p2(y1);
    p3(y1,y);
procedure p2(x2) $\text{RMOD}_{p2}\text{-}x2 = \text{RMOD}_{p1}\text{-}y1$
    p1(x2);
    p1(z);

Solution: $\text{RMOD}_{p3}\text{-}x3 = \text{RMOD}_{p3}\text{-}y3 = \text{RMOD}_{p1}\text{-}y1 = \text{RMOD}_{p2}\text{-}x2 = \text{true}$. 
Graphical way of solving RMOD equations: marker propagation

A reference parameter of a procedure \( f \) may be modified by execution of \( f \) if:
- \( f \) may write to parameter directly
- \( f \) passes it to procedure \( g \) as a reference parameter, and \( g \) may modify the parameter

RMOD computation

- build reverse of binding graph (ignore globals)
- if \( f \) modifies reference parameter \( r_1 \) directly, put a mark on \( r_1 \)
- propagate marks along reverse binding graph edges
- at the end of propagation, any variable that is marked corresponds to a reference parameter that may be modified
Side-effects to globals: similar to call-by-value

GLOBAL x, y, z;
procedure MAIN()
  p1(y);
  p1(x);
  p3(y, z);

procedure p3(x3, y3) GLOBALp3 = \{x\}
  x3 := \ldots;
  y3 := \ldots;
  x := \ldots;

procedure p1(y1) GLOBALp1 = GLOBALp2 U GLOBALp3
  p2(z);
  p2(y1);
  p3(y1, y);

procedure p2(x2) GLOBALp2 = GLOBALp1
  p1(x2);
  p1(z);

Solution: GLOBALp1 = GLOBALp2 = GLOBALp3 = \{x\}
From RMOD and GLOBALS, we can read off GMOD sets:

\[ \text{GMODp1}(m) = \{x,m\} \]
\[ \text{GMODp2}(m) = \{x,m\} \]
\[ \text{GMODp3}(m,n) = \{x,m,n\} \]

Oops...

What went wrong?
We did not take into account side-effects to globals that were passed as parameters![Kennedy and Cooper]

Correct equations for GLOBAL sets: use RMOD information for globals passed as parameters:

GLOBAL x,y,z;

procedure MAIN()
    p1(y);
p1(x);
p3(y,z);

procedure p3(x3,y3)  
    GLOBALp3 = {x}
x3 := ...;
y3 := ...;
x := ...;

procedure p1(y1)  
    GLOBALp1 = {z} U GLOBALp2 U GLOBALp3 U {y}
p2(z);
p2(y1);
p3(y1,y);

procedure p2(x2)  
    GLOBALp2 = {z} U GLOBALp1
p1(x2);
p1(z);

This gives the correct sets.
Algorithm for side-effects to globals: GLOBAL

• Find acyclic condensate of the call graph.
• For each procedure, determine set of globals either assigned to directly in procedure or passed by reference as a parameter to a procedure that modifies that parameter (use RMOD information for this).
• Union these sets for all procedures in an scc.
• Propagate these global sets in reverse invocation order in the acyclic condensate.
Putting it all together: GMOD computation

- **Compute RMOD information**
  - build binding graph
  - mark every node that represents a reference parameter modified directly by its procedure
  - propagate marks in binding graph: efficient approach would compute SCC’s and propagate in acyclic condensate
  - for each procedure $f$, read off $\text{RMOD-}f = \text{set of reference parameters of } f \text{ that are marked.}$

- **Compute GLOBALs information**
  - build call graph
  - for each procedure, find all globals that are either modified directly by procedure or passed as a reference parameter to another procedure that modifies that parameter (use RMOD for this)
• find scc’s of call graph and propagate sets in reverse invocation order.

• From RMOD and GLOBALs sets, read off GMOD function for each procedure f.
Summary

- Inter-procedural dataflow analysis: unknowns are functions on dataflow values
- Need to solve recursive functional equations
- Important special cases for which exact solution is possible
  - finite lattice: use tabular method
  - acyclic call graph: interpret equations as program
- Approximate solution of functional equations: approximate effect of backedges in call-graph
- Call-by-reference: need to take aliases into account
- Some interprocedural dataflow analysis problems can be reduced to marker propagation by formulating in the right graph. Key structure to exploit: strongly connected components in call graph.