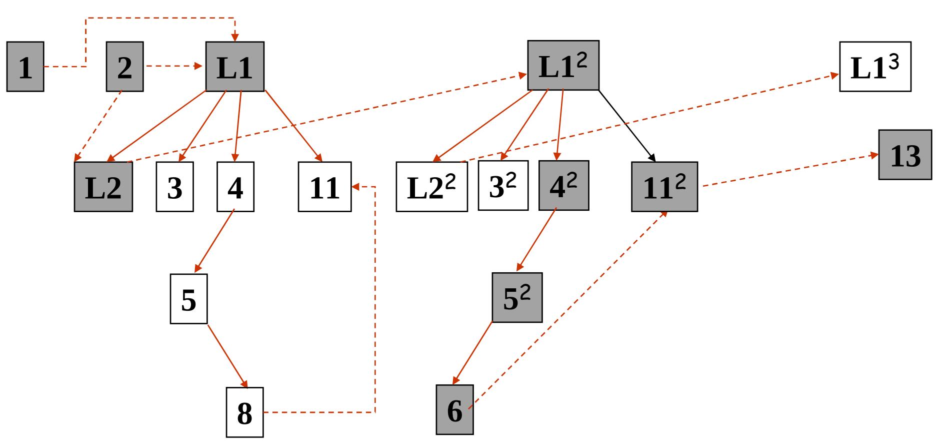
Class 8

- Review; questions
- Discuss Problem Set 4 questions
- Assign (see Schedule for links)
  - Complications of analysis—interprocedural control dependence, pointers, etc.
  - Problem Set 4: due 9/15/09

Dynamic Slicing Dependence Graphs-3

1^1, 2^1, 3^1, 4^1, 5^1, 8^1, 11^1, 2^2, 3^2, 4^2, 5^2, 6^1, 11^2, 2^3, 13^1



Program Slicing

1. Slicing overview
2. Types of slices, levels of slices
3. Methods for computing slices
4. Interprocedural slicing

Methods for Computing Slices

- **Data-flow on the flow graph**
  - Intraprocedural: control-flow graph (CFG)
  - Interprocedural: interprocedural control-flow graph (ICFG)
- **Reachability in a dependence graph**
  - Intraprocedural: program-dependence graph (PDG)
  - Interprocedural: system-dependence graph (SDG)
- **Information-flow relations**
  - Won't cover this method

Slicing Multi-procedures

```
int main() {
     int sum = 0;
     int i = 1;
     while (i < 11) {
          sum = add(sum,i);
          i = add(i,1);
     }
     printf(“%d\n”,sum);
     printf(“%d\n”,i);
}
```

```
int add(int x, int y) {
     return x + y;
}
```

Slicing Multi-procedures

```
int main() {
     int sum = 0;
     int i = 1;
     while (i < 11) {
          sum = add(sum,i);
          i = add(i,1);
     }
     printf(“%d\n”,sum);
     printf(“%d\n”,i);
}
```

```
int add(int x, int y) {
     return x + y;
}
```

Which statements actually affect the value of `i` at 10?

Slicing criterion: <10, i>

Slicing Multi-procedures

```
int main() {
     int sum = 0;
     int i = 1;
     while (i < 11) {
          sum = add(sum,i);
          i = add(i,1);
     }
     printf("%d\n",sum);
     printf("%d\n",i);
}
```

```
int add(int x, int y) {
     return x + y;
}
```

Slicing criterion: <10, i>

Slicing Multi-procedures

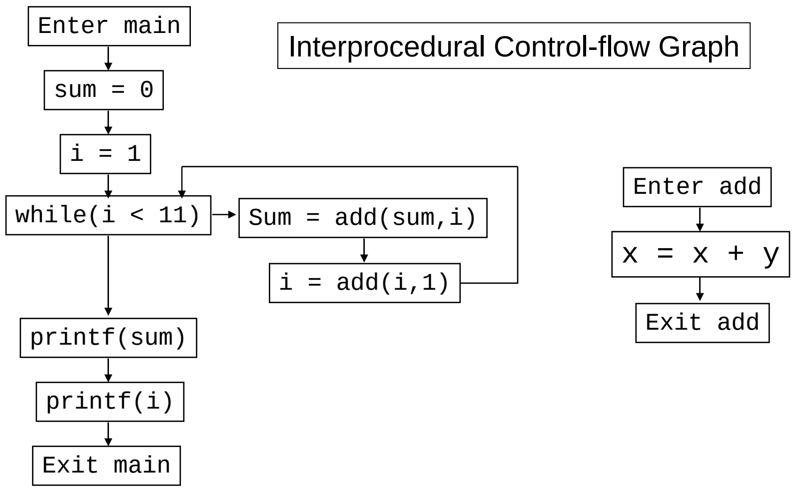
```
int main() {
     int sum = 0;
     int i = 1;
     while (i < 11) {
          sum = add(sum,i);
          i = add(i,1);
     }
     printf("%d\n",sum);
     printf("%d\n",i);
}
```

```
int add(int x, int y) {
     return x + y;
}
```

What does Weiser's algorithm compute for the slice for this criterion?

Slicing criterion: <10, i>

Slicing Multi-procedures



Slicing Multi-procedures

```
int main() {
      int sum = 0;
      int i = 1;
      while (i < 11) {
            sum = add(sum,i);
            i = add(i,1);
      }
      printf(“%d\n”,sum);
      printf(“%d\n”,i);
}
```

```
int add(int x, int y) {
      return x + y;
}
```

Results of applying Weiser’s algorithm

Slicing criterion: <10, i>

Interprocedural Dependences

Horwitz, Reps, Binkley: System Dependence Graph (SDG)

- Defined to address limitations of Weiser's technique
  - *Context-insensitivity:* main problem for interprocedural analysis of all kinds (e.g., control-flow, data-flow, control-dependence, slicing)
- Defined for a simplified language
  - Scalars, assignments, conditionals, while loops, returns, pass by copy-restore
  - Extensible to other languages (may later papers address extensions)
- SDG is a set of connected extended PDGs
  (Program/Procedure Dependence Graphs)
- Slicing is performed on the SDG
- May not compute executable slices

Extended PDGs for SDGs

Types of vertices in an extended PDG for procedure P

- Assignment statements
- Control predicates
- Entry vertex to P
- Formal-in parameters: represents initial definition of x for each x used before being defined in P
- Formal-out parameters: Final use of x for each x defined in P

Types of edges in extended PDG

- Control dependence
- Data dependence

Each call site to procedure Q is extended to have nodes for

- Call to Q
- Actual-in parameters and actual-out parameters for call to Q

New edges in extended PDG

- entry node to formal-in parameters (control-dependence)
- call node to actual-in parameters (control-dependence)

Connecting PDGs to Get SDG

New edges to connect extended PDGs to get SDG

- call node of P to entry nodes of those procedures it calls (call relation)
- actual parameters in P to formal parameters in those procedures it calls (data-dependence)

Procedure Calls, Parameter Passing

Goals for the representation of calls

- Modularity: build PDGs and then connect
- Simple connectivity: connect PDGs at call sites
- Efficiency and precision (of slicing): considers calling context
- Ease of parameter passing: Non-standard representation (i.e., copy-restore) for parameter passing (later extensions provided other methods for parameter passing)

Procedure Calls, Parameter Passing

```
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        add(sum,i);
        add(i,1);
    }
    printf("%d\n",sum);
    printf("%d\n",i);
```

```
add(int x, int y)
  {
    x = x + y;
     return;
}
```

1. Before the call, the calling procedure copies actual parameters to temporary values
2. Formal parameters of the called procedure are initialized using the corresponding temporary values
3. Before the return, the called procedure copies the final values of the formal parameters to the temporary variables
4. After returning, the calling procedure updates the actual parameters by copying the values of the corresponding temporary variables

Procedure Calls, Parameter Passing

```
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
        add(sum,i);
        xin = sum;
        yin = i;
        call add;
```

```
add(int x, int y)
  {
    x = x + y;
     return;
}
```

1. **Before the call**, the calling procedure copies actual parameters to temporary values
2. Formal parameters of the called procedure are initialized using the corresponding temporary values
3. Before the return, the called procedure copies the final values of the formal parameters to the temporary variables
4. After returning, the calling procedure updates the actual parameters by copying the values of the corresponding temporary variables

Procedure Calls, Parameter Passing

```
1.int main() {
2.   int sum = 0;
3.   int i = 1;
4.   while (i < 11) {
5.          add(sum,i);
            xin = sum;
            yin = i;
            call add;
```

```
11.add(int x, int y)
   { x = xin;
     y = yin;
12.  x = x + y;
13.   return;
14.}
```

1. Before the call, the calling procedure copies actual parameters to temporary values
2. **Formal parameters** of the called procedure are initialized using the corresponding temporary values
3. Before the return, the called procedure copies the final values of the formal parameters to the temporary variables
4. After returning, the calling procedure updates the actual parameters by copying the values of the corresponding temporary variables

Procedure Calls, Parameter Passing

```
1.int main() {
2.   int sum = 0;
3.   int i = 1;
4.   while (i < 11) {
5.          add(sum,i);
            xin = sum;
            yin = i;
            call add;
```

```
11.add(int x, int y)
   { x = xin;
     y = yin;
12.  x = x + y;
      xout = x;
      yout = y;
13.   return;
14.}
```

1. Before the call, the calling procedure copies actual parameters to temporary values
2. Formal parameters of the called procedure are initialized using the corresponding temporary values
3. **Before the return**, the called procedure copies the final values of the formal parameters to the temporary variables
4. After returning, the calling procedure updates the actual parameters by copying the values of the corresponding temporary variables

meter Passing

```
int main() {
    int sum = 0;
    int i = 1;
    while (i < 11) {
          add(sum,i);
          xin = sum;
          yin = I
          call add;
          sum = xout;
          i = yout;
```

```
add(int x, int y)
  { x = xin;
    y = yin;
    x = x + y;
    xout = x;
    yout = y;
    return;
}
```

1. Before the call, the calling procedure copies actual parameters to temporary values
2. Formal parameters of the called procedure are initialized using the corresponding temporary values
3. Before the return, the called procedure copies the final values of the formal parameters to the temporary variables
4. **After returning**, the calling procedure updates the actual parameters by copying the values of the corresponding temporary variables

Procedure Calls, Parameter Passing

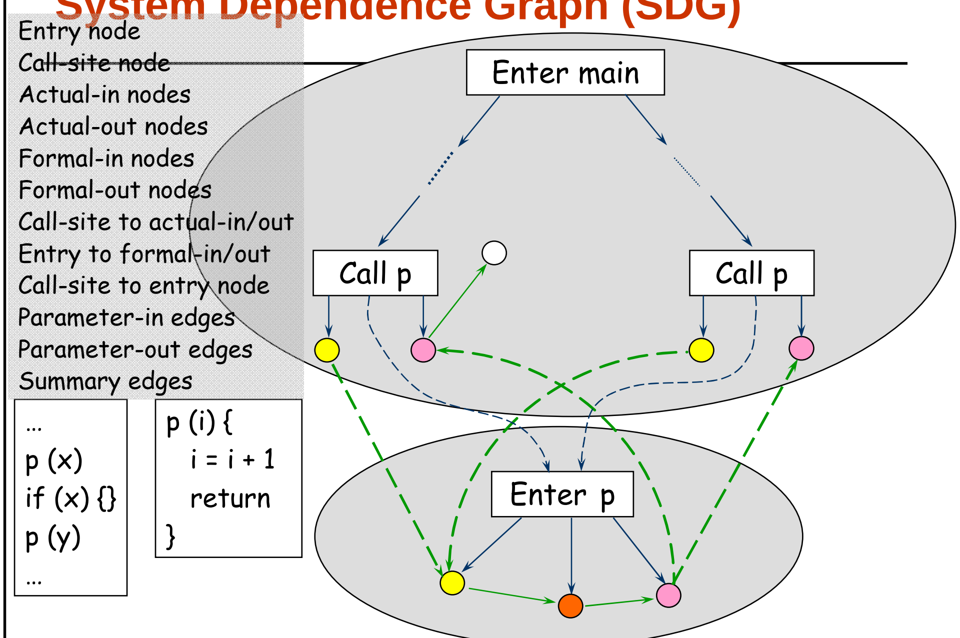
- Each PDG is extended to have nodes for procedure parameters and function result
  - Entry node
  - Formal-in nodes
  - Formal-out nodes
- Each call statement is extended with
  - Call-site node
  - Actual-in nodes
  - Actual-out nodes
- Appropriate edges (intra and inter)
  - Call-site node to actual-in/out (control-dependence)
  - Entry node to formal-in/out (control-dependence)
  - Call-site node to entry node (control dependence)
  - Parameter-in edges, from actual-in to formal-in (data-dependence)
  - Parameter-out edges, from formal-out to actual-out (data-dependence)
  - Summary edges, between formal in and formal out (data-dependence)

Procedure Calls, Parameter Passing

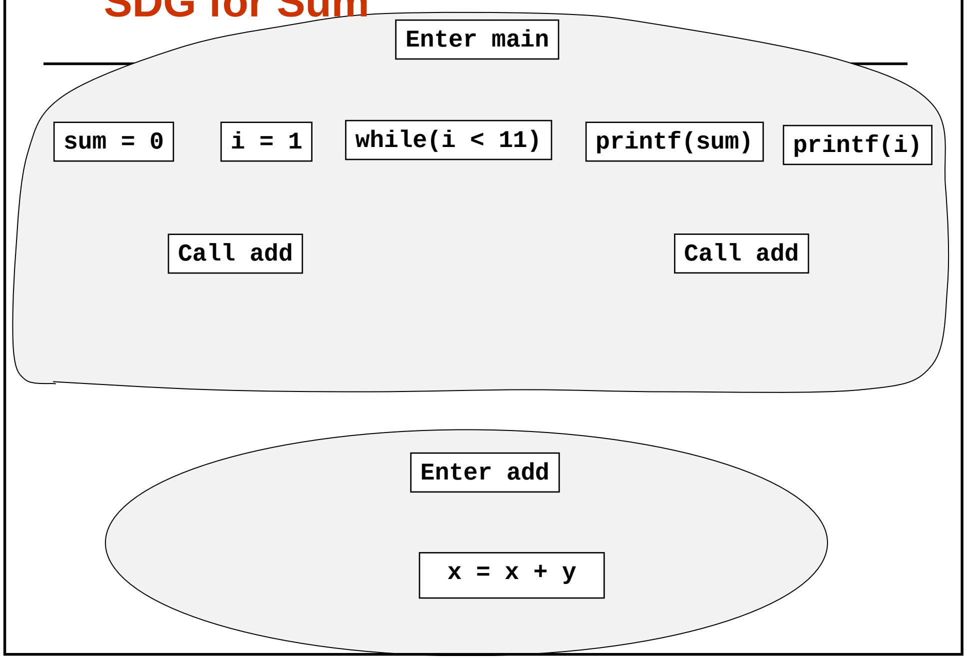
How do we decide which values are transferred in and out?

- All actual parameters are copied in and out
  - For each actual parameter x (for a formal parameter r) in a call p → q
    - One actual-in "r_{in} = x"
    - If x is a variable, one actual-out "x = r_{out}"
  - For each formal parameter r in a call p → q
    - One formal-in "r = r_{in}"
    - One formal-out "r_{out} = r"
- We can be more precise than this, though. **How?**

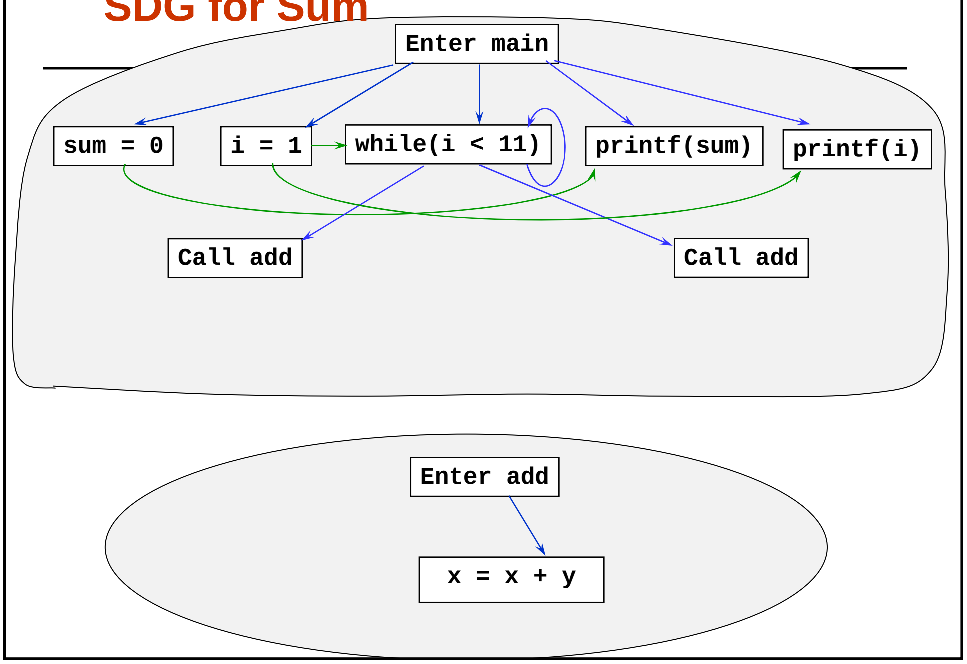
System Dependence Graph (SDG)



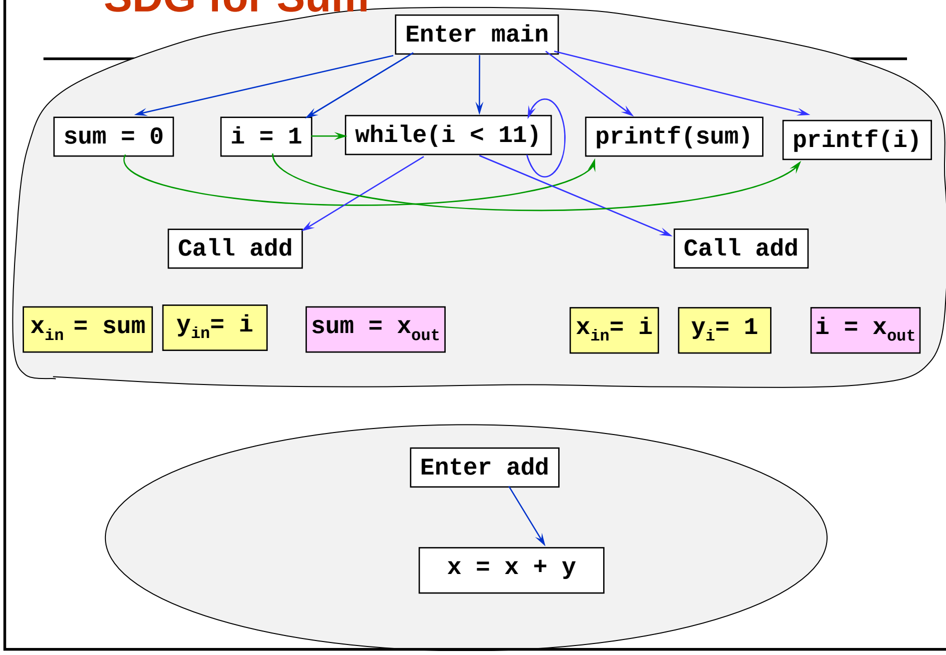
SDG for Sum



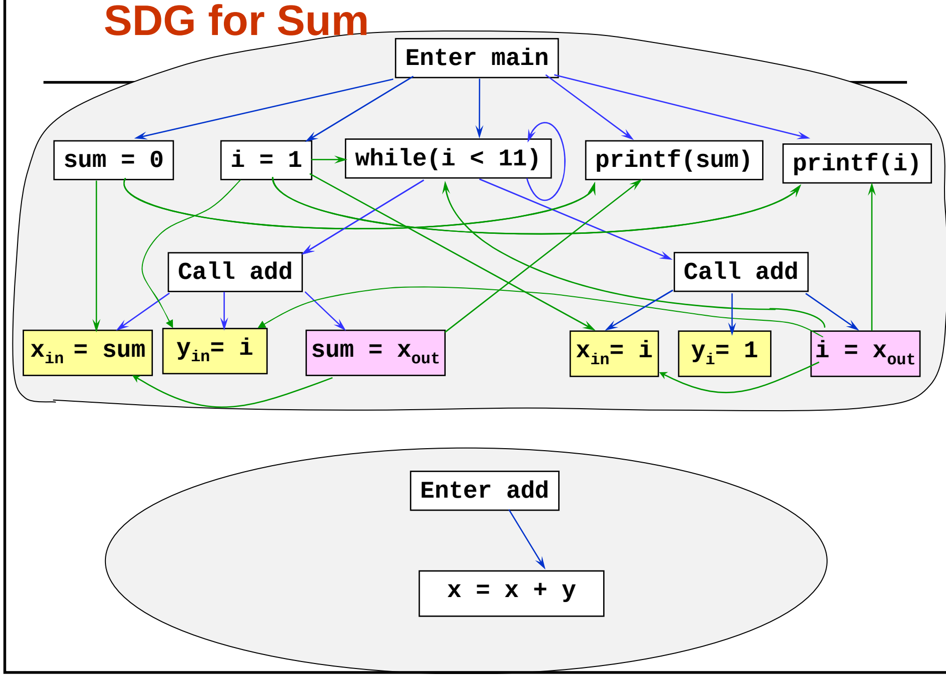
SDG for Sum



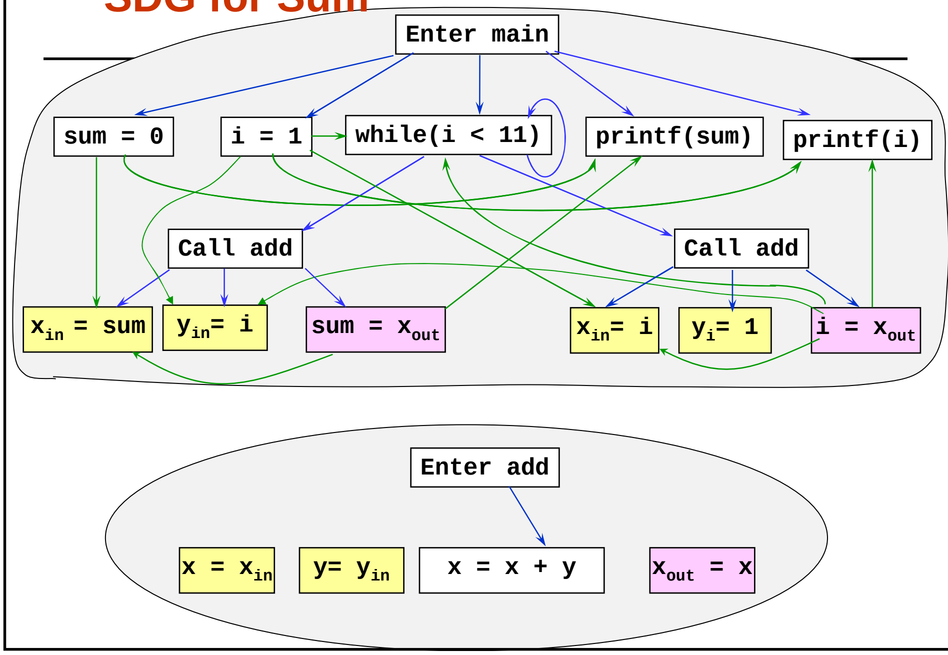
SDG for Sum



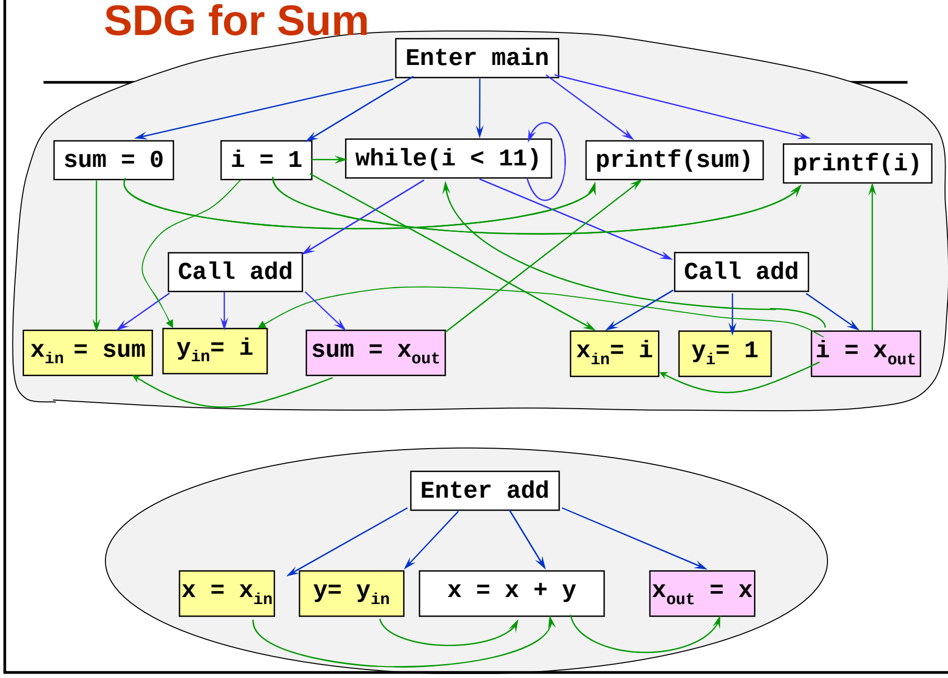
SDG for Sum



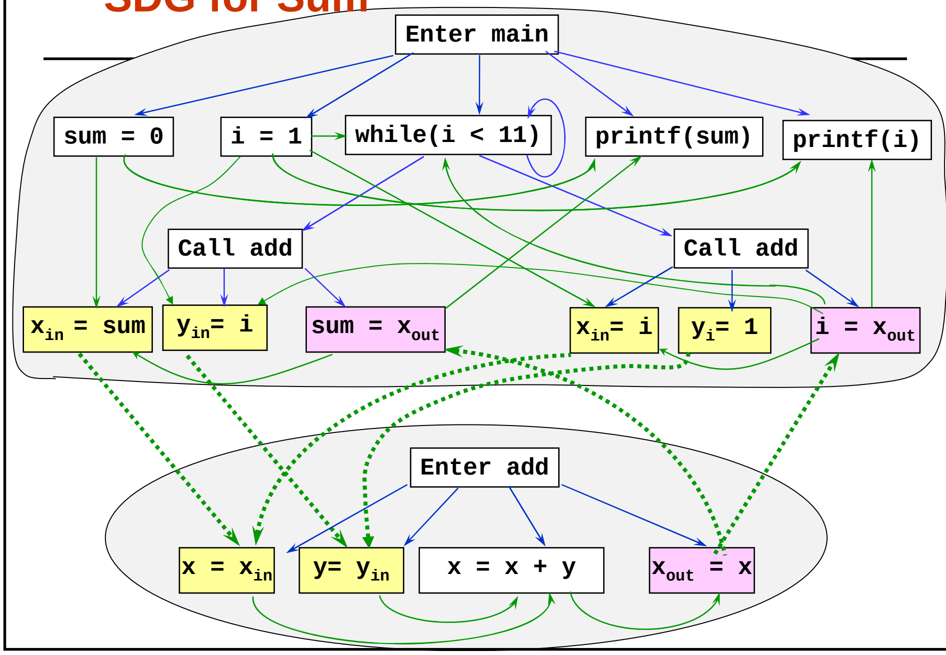
SDG for Sum



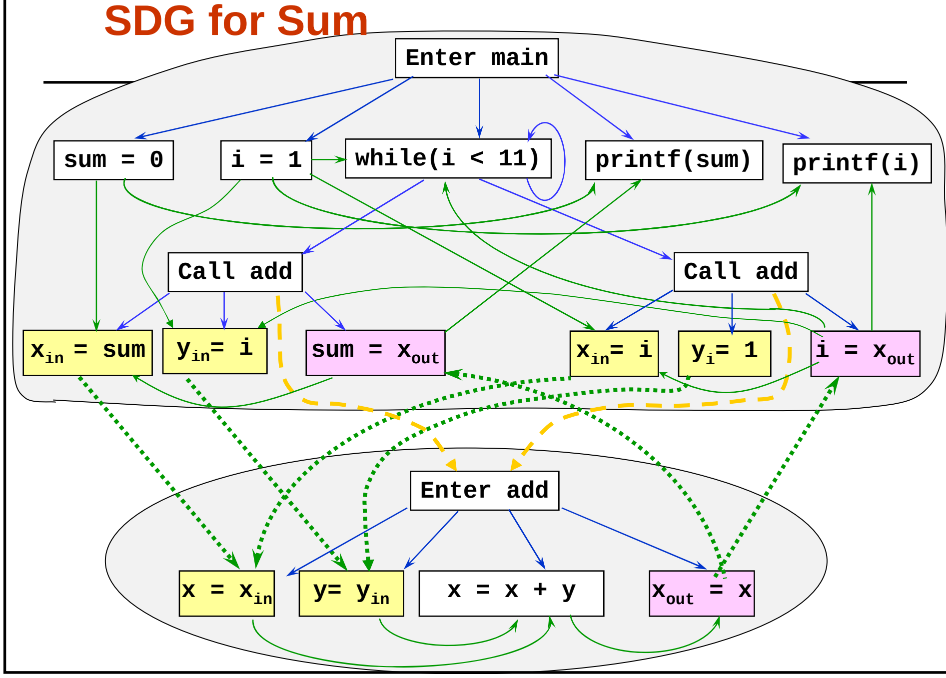
SDG for Sum



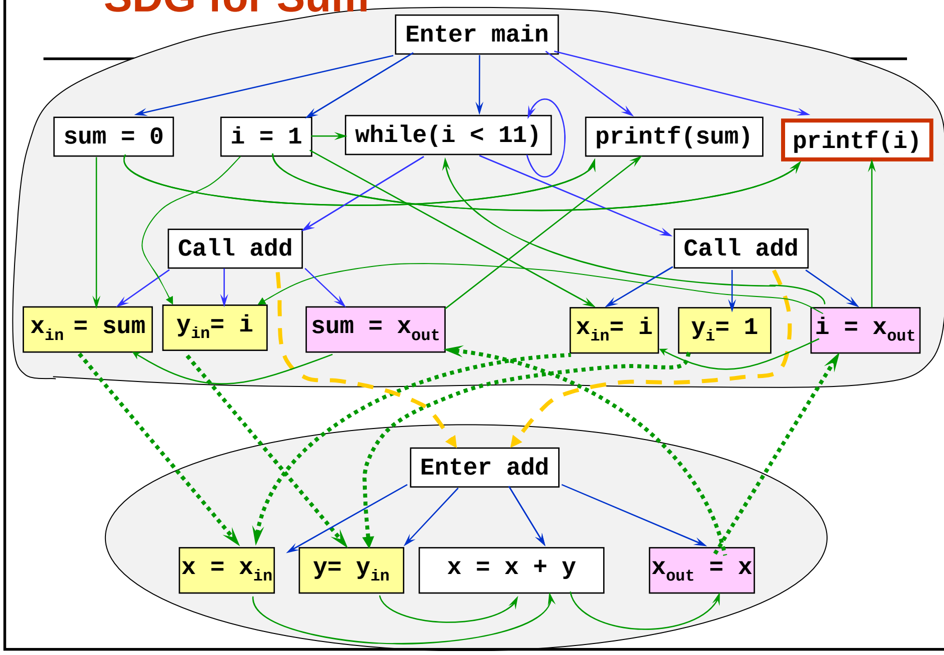
SDG for Sum



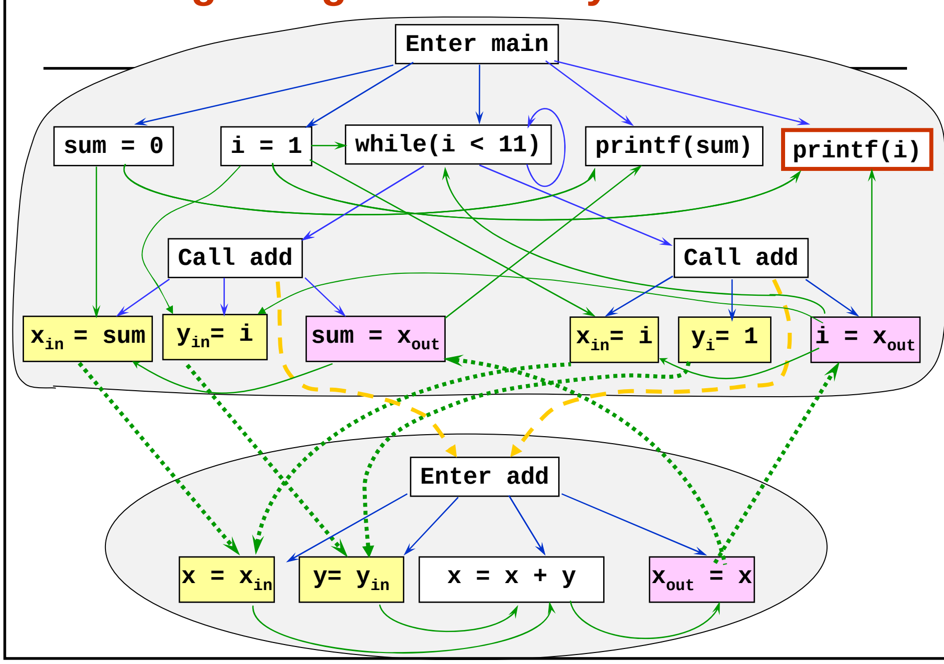
SDG for Sum



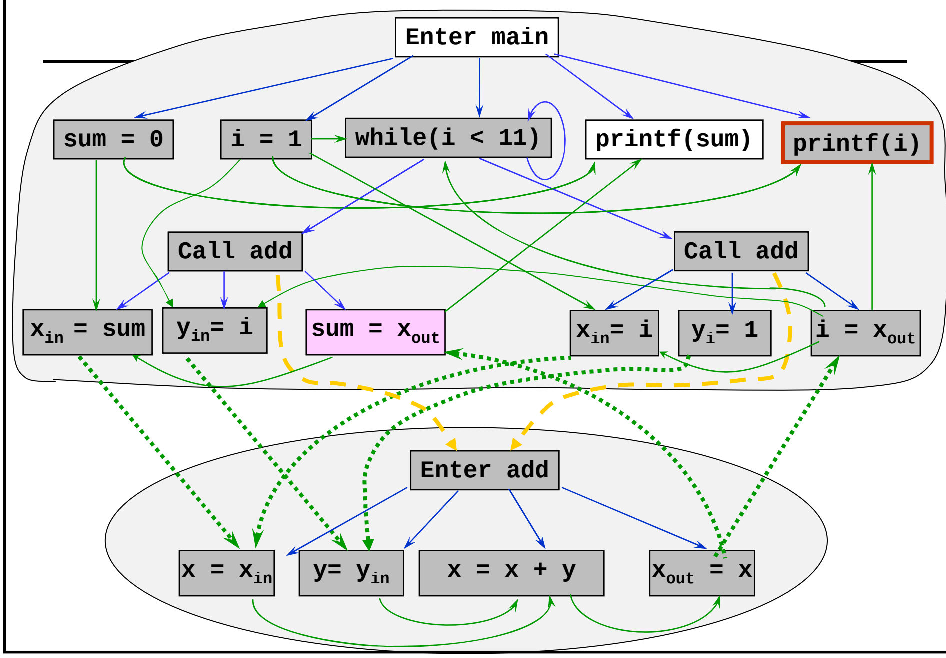
SDG for Sum



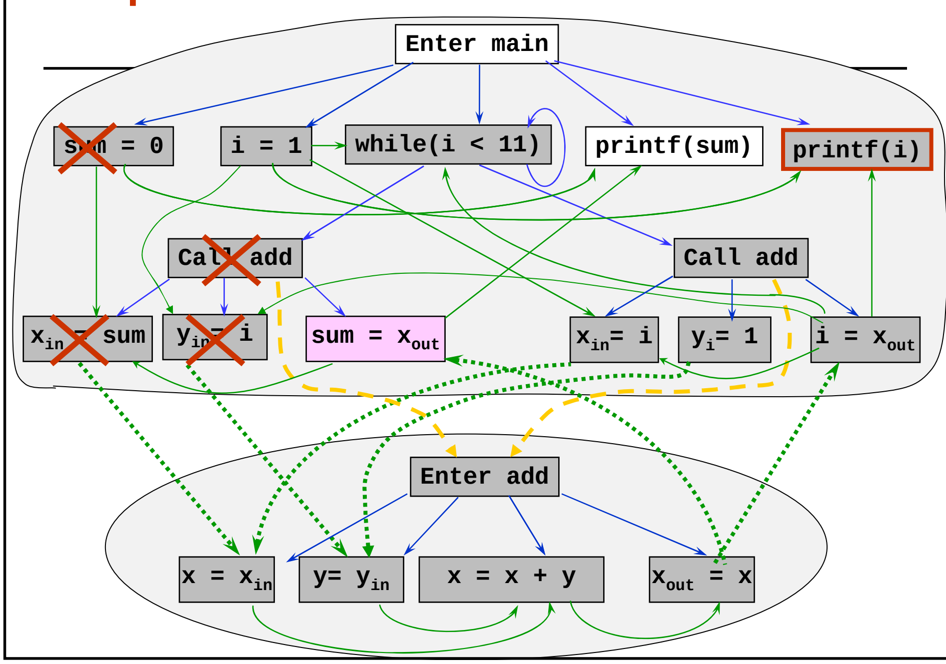
Slicing Using Reachability



Slicing Using Reachability



Imprecision



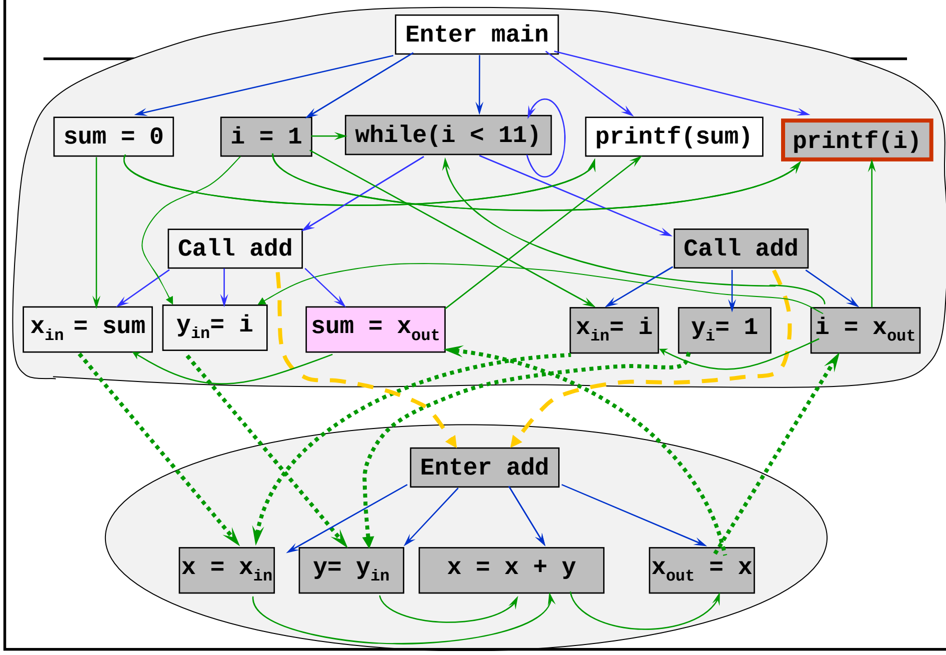
Precise Interprocedural Slicing

- What are some solutions?

Precise Interprocedural Slicing

- Match procedure returns with the corresponding calls when traversing SDG

Precise Interprocedural Slicing

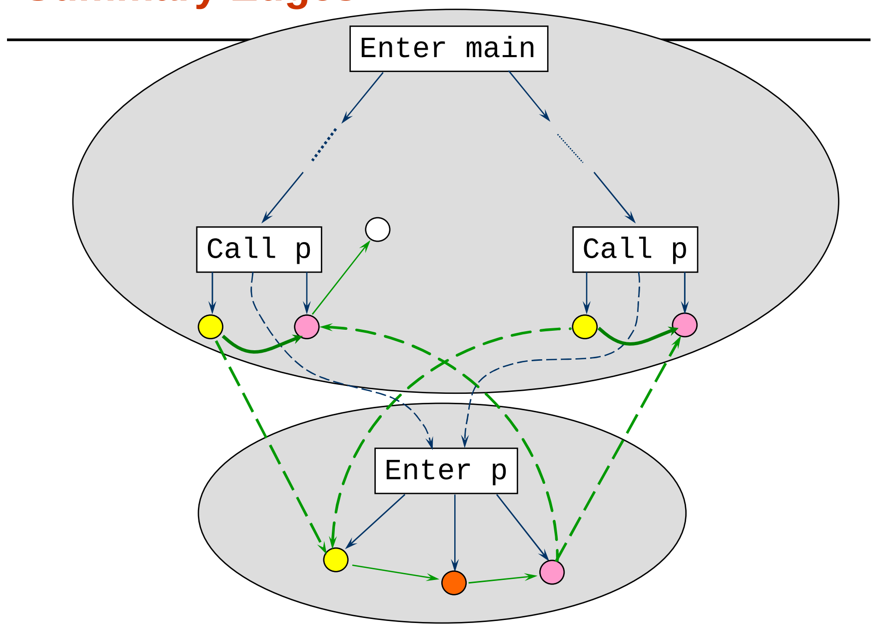


Two-phase Reachability Slicing Algorithm

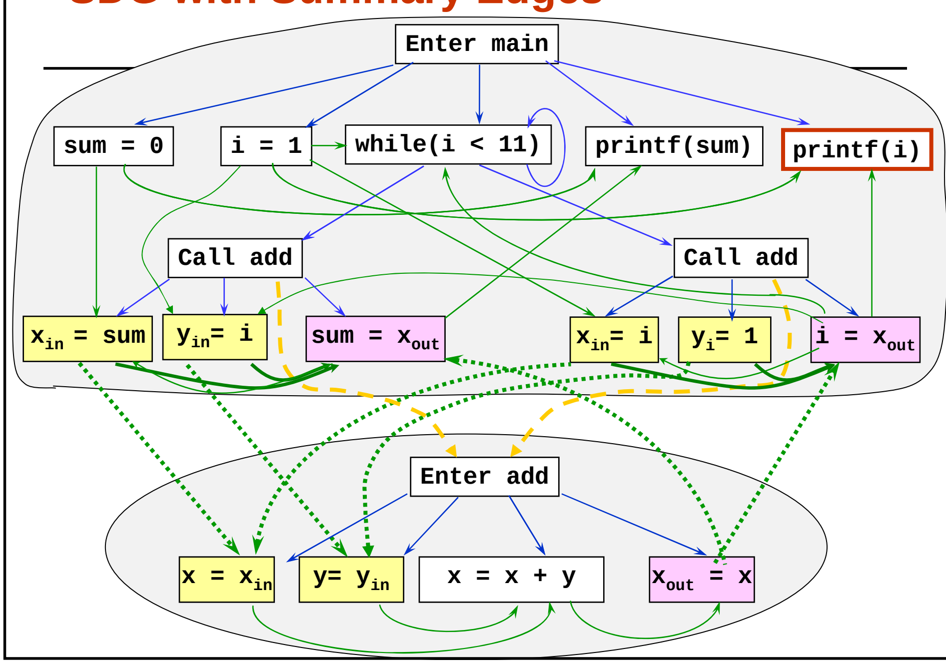
To avoid the mismatches of procedure returns and procedure calls when traversing the graph

- Phase I: find the statements in the current procedure and the callers of the current procedure that may affect the slicing criterion
  - Do not traverse return edges
  - Use summary information to continue the slicing at each callsite
- Phase II: Find the statements in the callees of the current procedure that may affect the slicing criterion
  - Do not traverse call edges

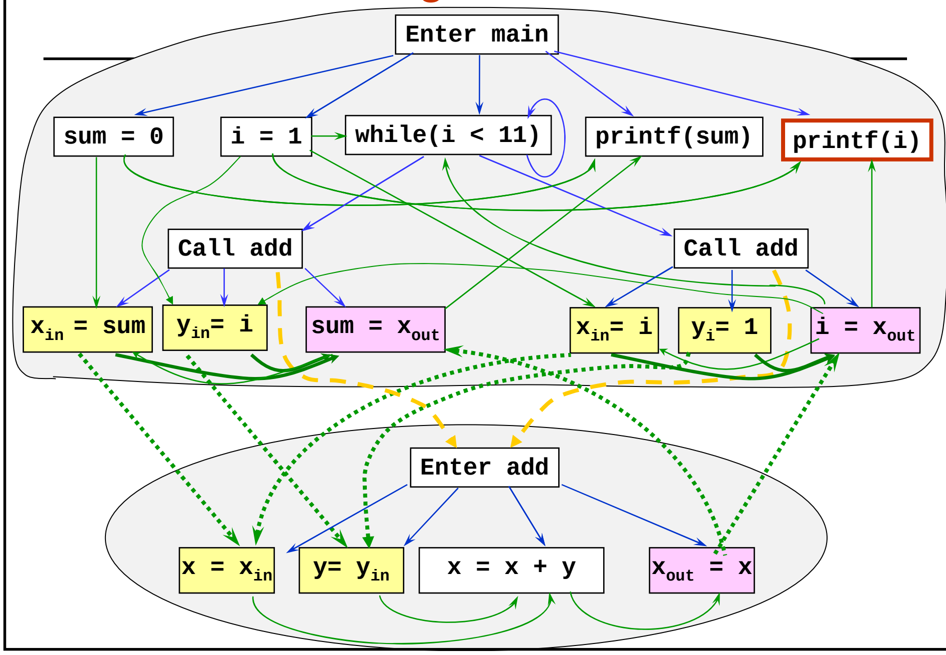
Summary Edges



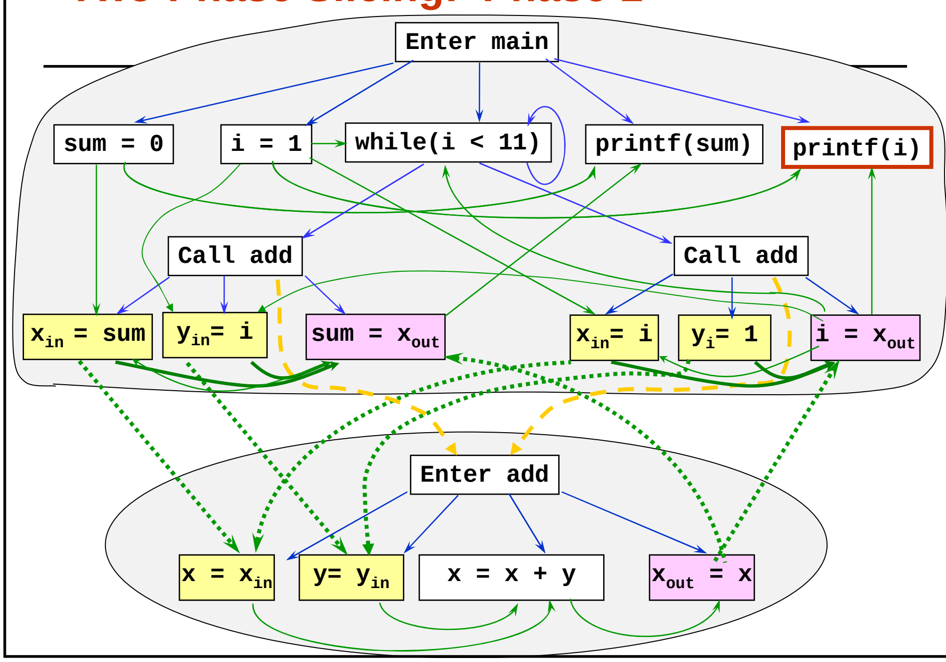
SDG with Summary Edges



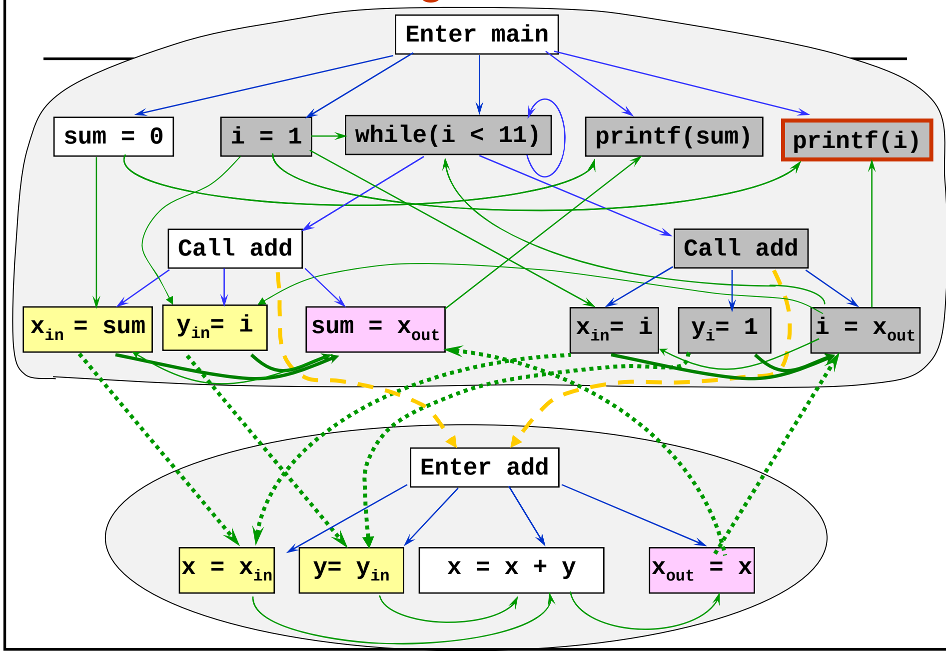
Two-Phase Slicing



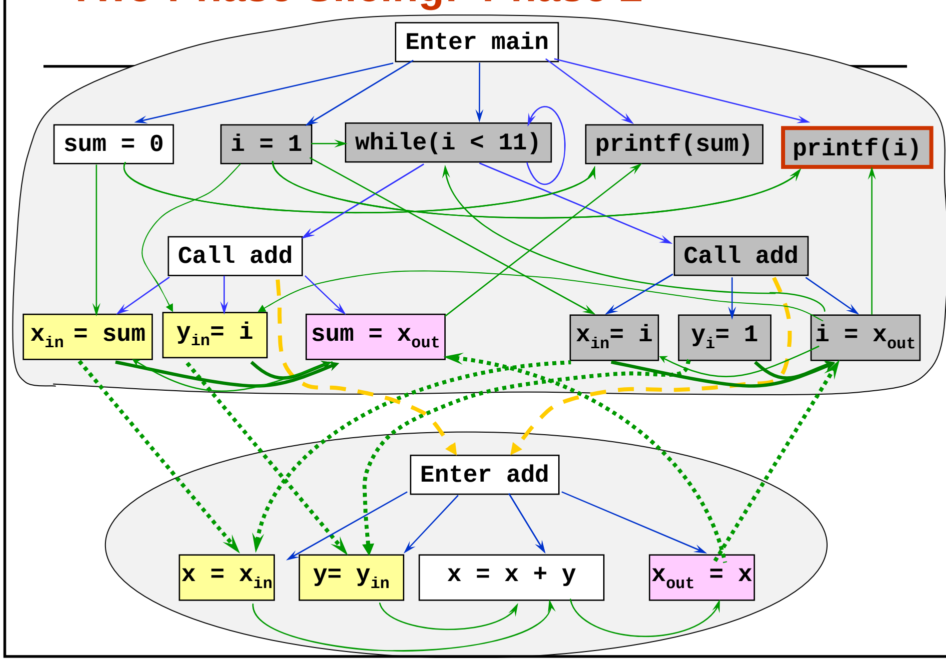
Two-Phase Slicing: Phase 1



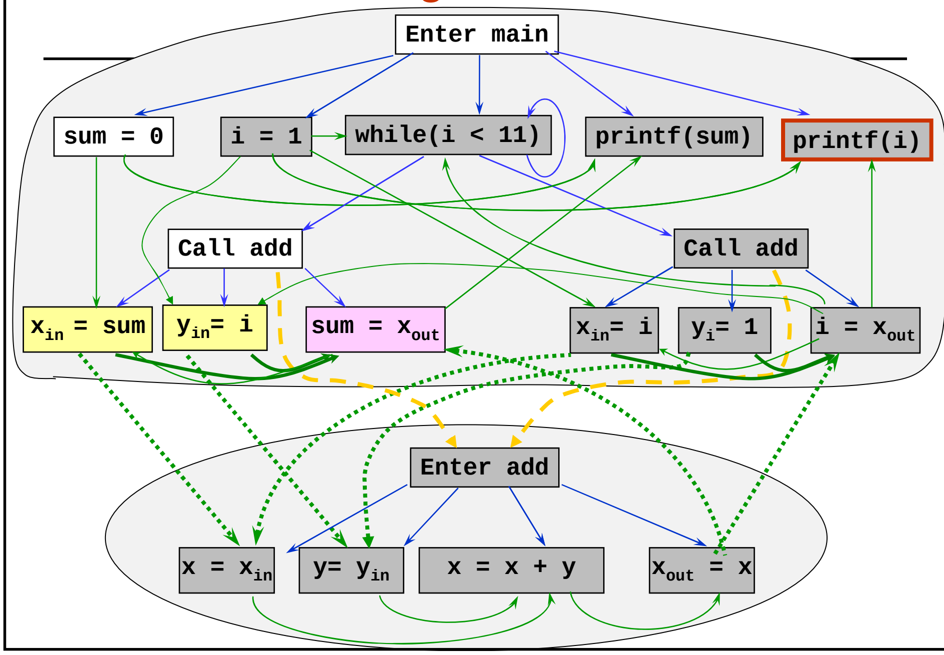
Two-Phase Slicing: Phase 1



Two-Phase Slicing: Phase 2



Two-Phase Slicing: Phase 2



Iterative Computation of the Summary Edges

Step 1: compute the reachability from formal-in nodes to formal-out nodes in each procedure

Step 2: create the summary edges in each caller according to the reachability from formal-in nodes to formal-out nodes in a procedure

Step 3: update the reachability from formal-in nodes to formal-out nodes of each caller

Step 4: if Step 3 produces new results, go to step 2