

Inter-Procedural Data Flow Analysis

Giovanni Denaro

giovanni.denaro@unimib.it

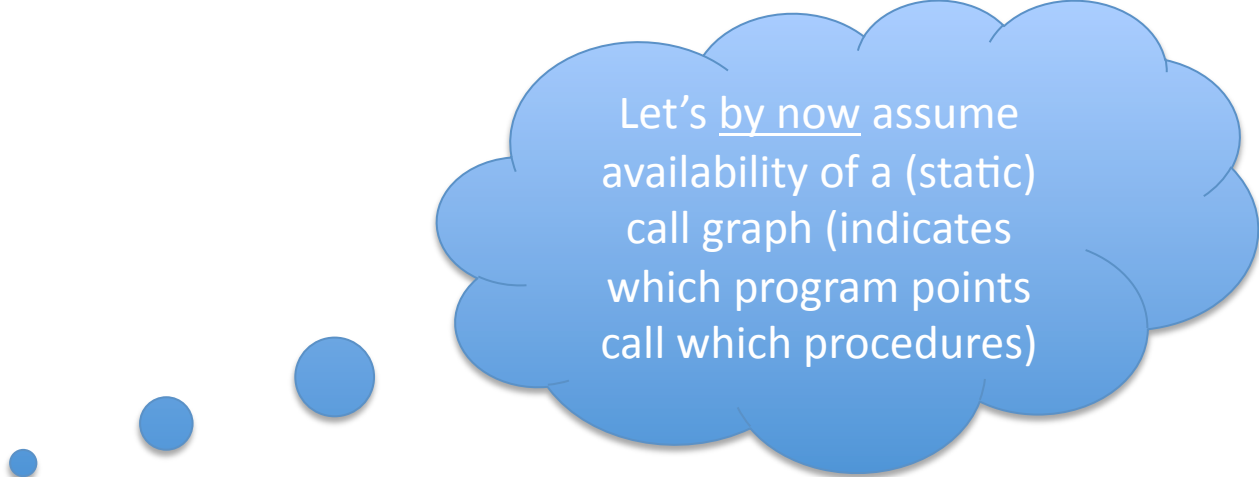
Agenda

- Inter-procedural data flow analysis
- Context sensitivity
- Impact of pointers and aliases
- Data flow analysis of OO programs

Scope of data flow analysis

- Intra-procedural
 - Within a single method or procedure
 - as described so far
- Inter-procedural
 - Across several methods (and classes) or procedures
 - Uses calling relationships among procedures
- Cost/Precision trade-offs for inter-procedural analysis are critical, and difficult
 - context sensitivity
 - impact of pointers and aliases

Inter-procedural analysis



Let's by now assume availability of a (static) call graph (indicates which program points call which procedures)

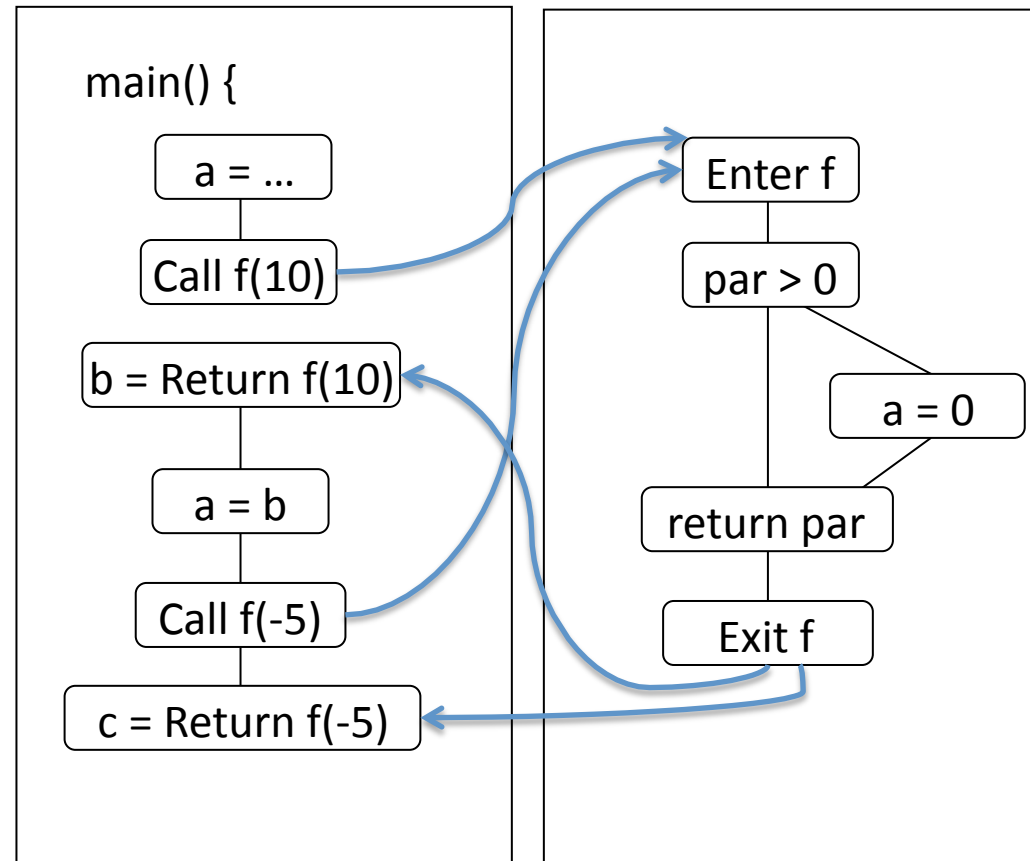
- Naive approach
 - Go through data flow analysis on the $I^{\text{Interproc}}$ CFG
 - ICFG combines combines the call graph and the CFGs of all procedures

(Naive) Inter-procedural analysis

$Reach(n) = \cup ReachOut(m) \mid m \in pred(n)$
 $ReachOut(n) = (Reach(n) \setminus kill(n)) \cup gen(n)$

$gen(n) = \{ v_n \mid v \text{ defined at } n \}$
 $kill(n) = \{ v_x \mid v \text{ defined at } x \neq n \}$

```
1. int a, b, c;
2. main() {
3.   a = ... ; //input from user
4.   b = f(10);
5.   a = b;
6.   c = f(-5);
7. }
8. f(int par) {
9.   if(par > 0)
10.    a = 0;
11.   return par;
12. }
```

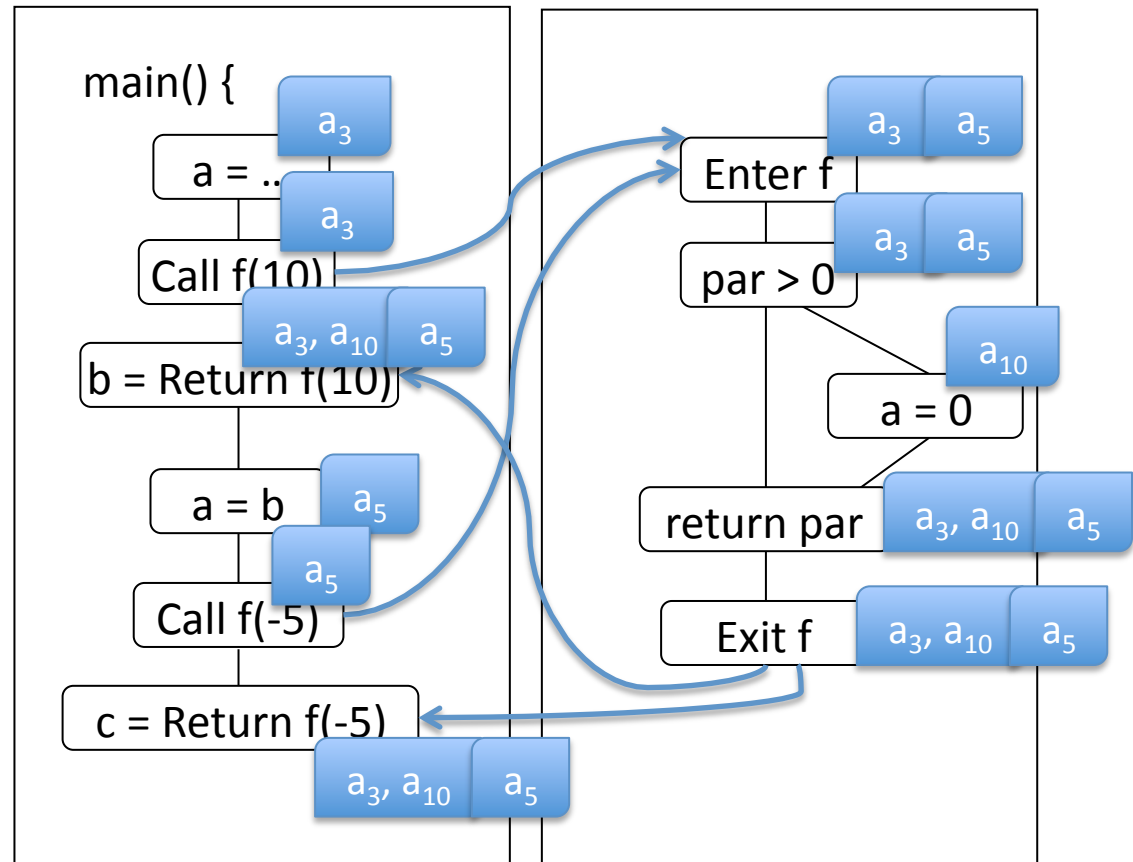


(Naive) Inter-procedural analysis

$Reach(n) = \cup ReachOut(m) \mid m \in pred(n)$
 $ReachOut(n) = (Reach(n) \setminus kill(n)) \cup gen(n)$

$gen(n) = \{ v_n \mid v \text{ defined at } n \}$
 $kill(n) = \{ v_x \mid v \text{ defined at } x \neq n \text{ AND at } n \}$

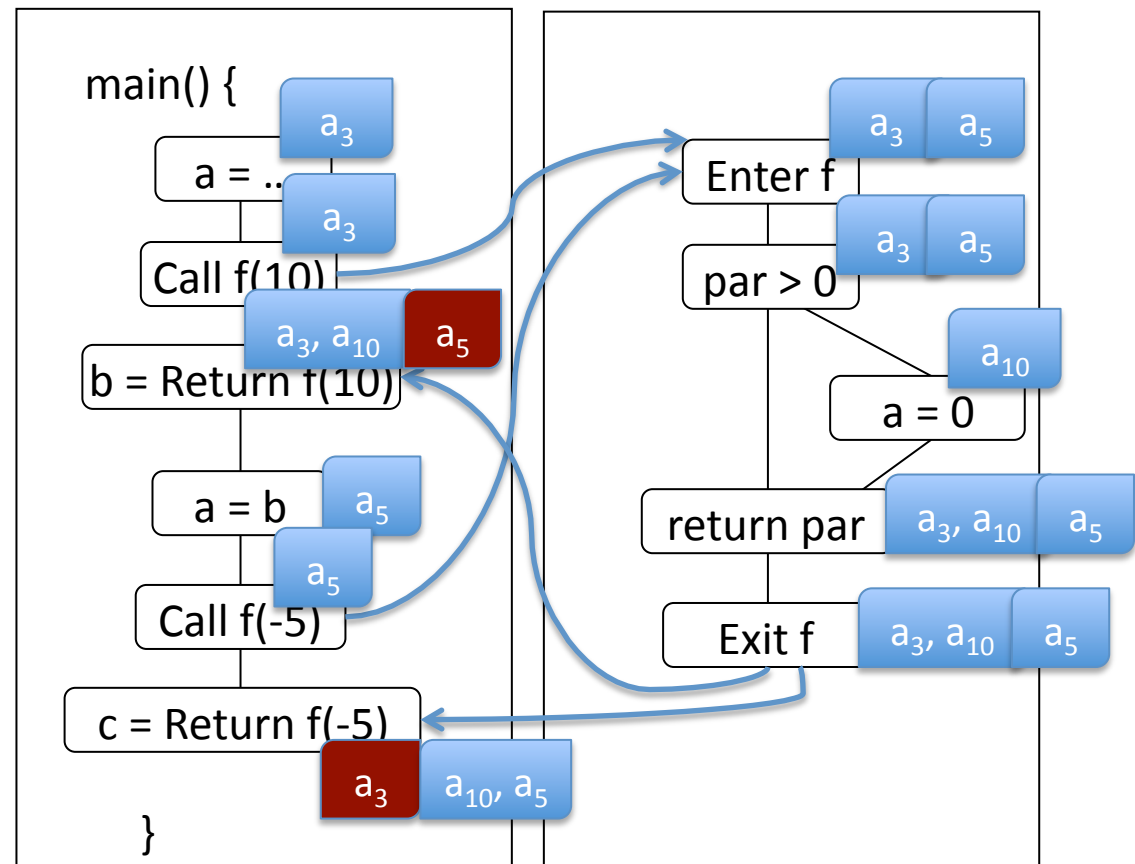
```
1. int a, b, c;
2. main() {
3.   a = ... ; //input from user
4.   b = f(10);
5.   a = b;
6.   c = f(-5);
7. }
8. f(int par) {
9.   if(par > 0)
10.    a = 0;
11.   return par;
12. }
```



(Naive) Inter-procedural analysis

Data flow facts from a call site can “pollute” results at other call sites
(it can be worse if we consider also the defs of b and c)

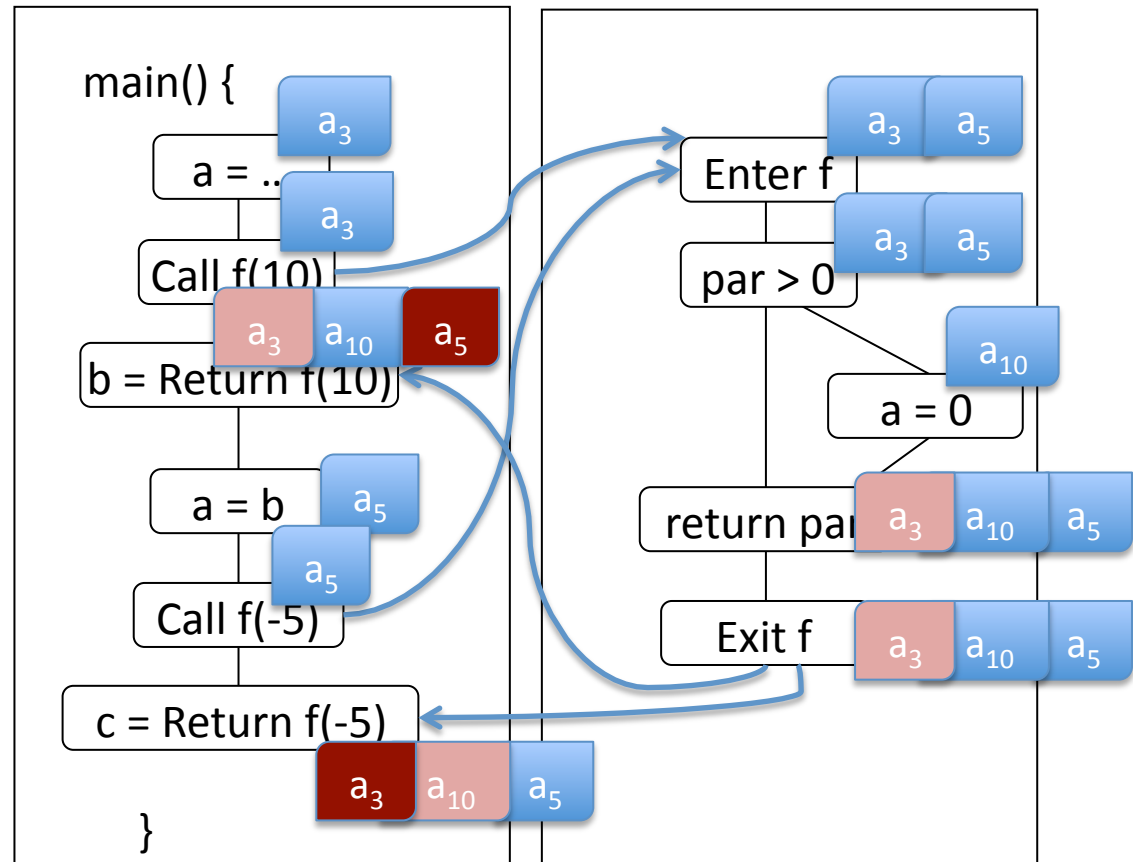
```
1. int a, b, c;
2. main() {
3.   a = ... ; //input from user
4.   b = f(10);
5.   a = b;
6.   c = f(-5);
7. }
8. f(int par) {
9.   if(par > 0)
10.    a = 0;
11.   return par;
12. }
```



(Naive) Inter-procedural analysis

Hinders call-site related optimization (e.g., by constant propagation)

```
1. int a, b, c;  
2. main() {  
3.   a = ... ; //input from user  
4.   b = f(10);  
5.   a = b;  
6.   c = f(-5);  
7. }  
  
8. f(int par) {  
9.   if(par > 0)  
10.    a = 0;  
11.   return par;  
12. }
```



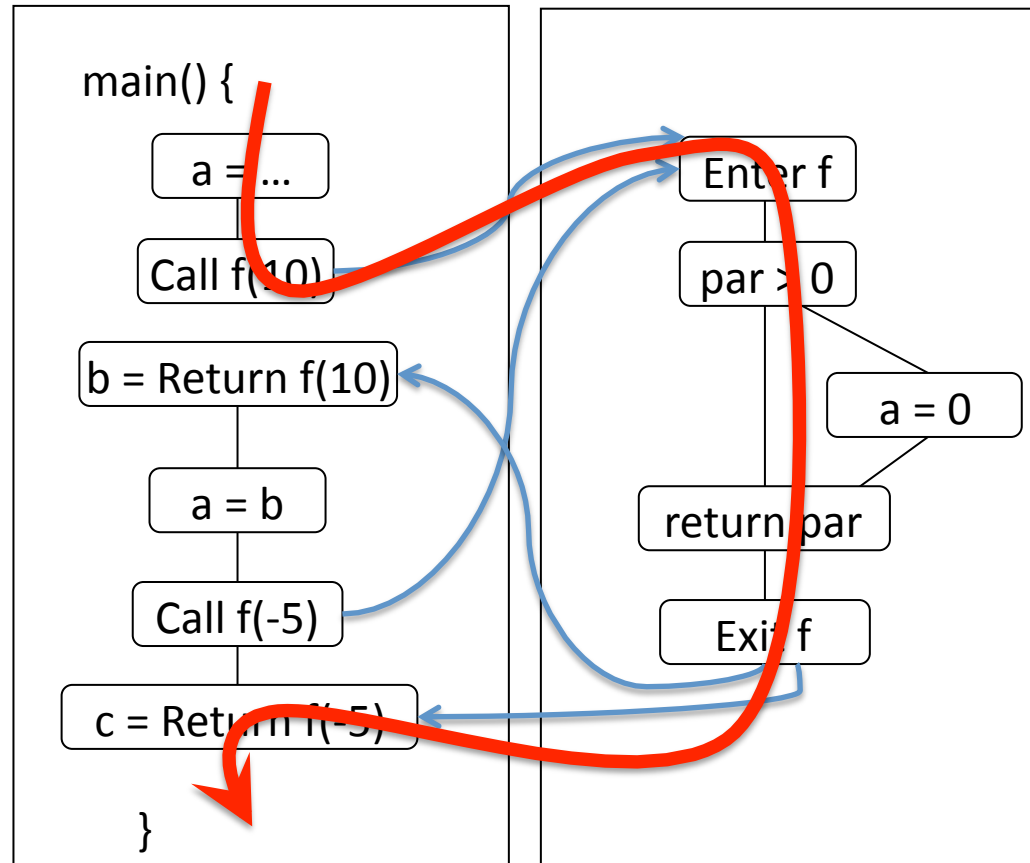
Inter-procedural analysis

In general

- Only some paths of the call graph are valid path according to the call-return semantics of the program
 - technical term: “realizable” paths
- The call graph can be not statically unavailable
 - E.g., think of OO languages that provide dynamic binding

A non-realizable path

```
1. int a, b, c;  
2. main() {  
3.   a = ... ; //input from user  
4.   b = f(10);  
5.   a = b;  
6.   c = f(-5);  
7. }  
  
8. f(int par) {  
9.   if(par > 0)  
10.     a = 0;  
11.   return par;  
12. }
```



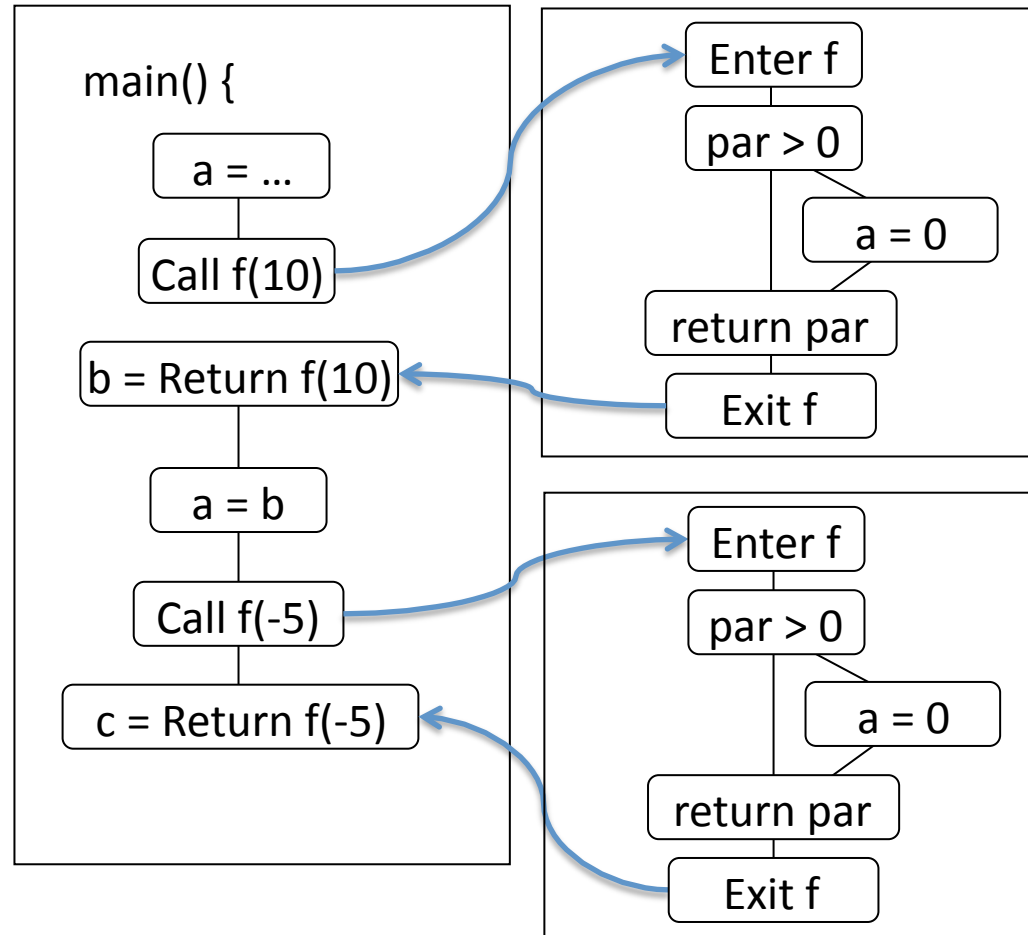
Context sensitivity

- Accounting for realizable paths by preventing propagation of data flow facts along inadmissible call-return sequences
- Context sensitive analyses
 - Distinguish “different” calls of the same procedure (possibly making a finite number of copies of procedures)
 - Use invocation context (the call site) information to determine the correct propagation of data flow paths (may include making the decision on when (not) to share a copy)
- The choice on what the context information is produces different precision-scalability tradeoffs
 - A common choice: approximation of the call chains
 - ...However we need to:
 - Bound the combinatorial number of procedure copies to be considered
 - Handle recursion

Context sensitivity by “copies”

```
1. int a, b, c;
2. main() {
3.   a = ... ; //input from user
4.   b = f(10);
5.   a = b;
6.   c = f(-5);
7. }

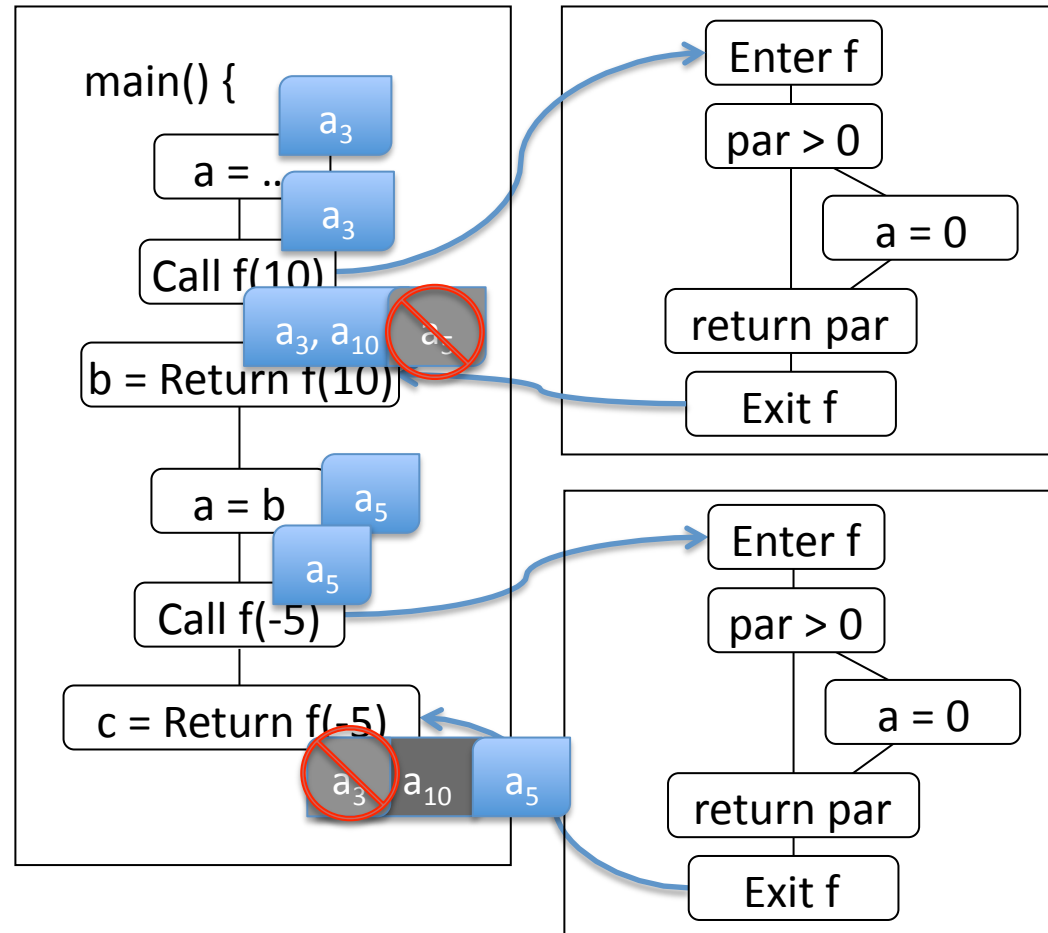
8. f(int par) {
9.   if(par > 0)
10.    a = 0;
11.   return par;
12. }
```



Context sensitivity

1. int a, b, c;
2. main() {
3. a = ... ; //input from user
4. b = f(10);
5. a = b;
6. c = f(-5);
7. }

8. f(int par) {
9. if(par > 0)
10. a = 0;
11. return par;
12. }



Now: a₅, a₃ propagate to the right statements only, a₁₀ (last) could be excluded by const-propagation, ...

Algorithm: Reps, Horwitz, Sagiv. POPL 1995

- MOP: Meet Over all-realizable Paths
 - Inter-procedural analysis as a graph reachability problem
 - Precise solution if
 - The set of data flow facts is **finite**
 - The dataflow propagation functions
 - facts entering an edge \rightarrow facts propagated through edge
 - are **distributive** wrt the meet operator
- We will look into this algorithm as exercise
 - based on the implementation available in IBM-WALA
 - and the RHS POPL'95 paper

Algorithm: Harrold, Soffa. TOPLAS 1994

1. Intra-procedural analysis of each procedure
 - Tracks propagation of formal/actual parameters to and from call sites
 - Objective: express the link between local data and data at procedure calls
 - Initial approximation: Definitions/uses **unpreserved** over procedure calls
2. Build the Inter-procedural Flow Graph + Inter-reaching edges
 - IFG: An edge from each actual to formal parameters and back of each call (call-entry and exit-return edges)
 - Inter-reaching edges: edges from call-nodes to return-nodes, abstract which defs/uses may traverse the callee and be preserved
 - The algorithm processes the IFG iteratively to compute inter-reaching edges
3. Perform data flow analysis on the IFG to obtain the inter-procedural information



Context information

Impact of pointers and aliases

Pointers/aliases can cause subtle and complex data dependencies

- Pointer accesses shall be reconciled with the accessed variables

A POSSIBLE CLASSIFICATION

Accesses to program variables

Direct : that is, with no pointer involved

Indirect : that is, through a pointer deference

Indirect accesses

Single-alias : points to single memory location

Multiple-alias : points to multiple locations

Impact of pointers and aliases

Pointers/aliases can cause subtle and complex data dependencies

- Pointer accesses shall be reconciled with the accessed variables
- Aliasing with multiple variables can
 - lead to further weaken the “may-ness” of data-flow analysis
 - foster multiple (and non-trivially computable) callees per call-site with programming languages that support dynamic binding

A POSSIBLE CLASSIFICATION

Accesses to program variables

Direct : that is, with no pointer involved

Indirect : that is, through a pointer deference

Indirect accesses

Single-alias : points to single memory location

Multiple-alias : points to multiple locations

```
proc(int cond1, int cond2) {  
    int x, y, z, *p;  
    z = 17;  
    x = 13;  
    if (cond1) {  
        p = &y;  
        *p = z;  
    }  
    else {  
        if (cond2)  
            p = &x;  
        else  
            p = &z;  
        *p = 7 + z;  
        y = 53;  
        p = &x;  
    }  
    x = x + y + z;  
    *p = *p + 5;  
    y = x + y;  
    ...  
}
```

Definition: Single alias access to y

Question: Is this a possible pair? →

Definition: Multiple alias access to x or z

Use: Direct access to y

Classification of data dependencies that depend on pointer accesses (i/ii)

Type of data dep.	Classification	Characterization
Definition/Use	Definite definition/use	Direct or single-alias access
	Possible definition/use	Multiple alias access
Def-clear path	Definite def-clear path	No definite re-definition No possible re-definition
	Possible def-clear path	No definite re-definition Some possible re-definition
	Definite Killing path	At least a definite re-definition

```
proc(int cond1, int cond2) {  
    int x, y, z, *p;  
    z = 17;  
    x = 13;  
    if (cond1) {  
        p = &y;  
        *p = z;  
    }  
    else {  
        if (cond2)  
            p = &x;  
        else  
            p = &z;  
        *p = 7 + z;  
        y = 53;  
        p = &x;  
    }  
    x = x + y + z;  
    *p = *p + 5;  
    y = x + y;  
    ...  
}
```

Definite definition of y

Possible definition of x
Possible definition of z

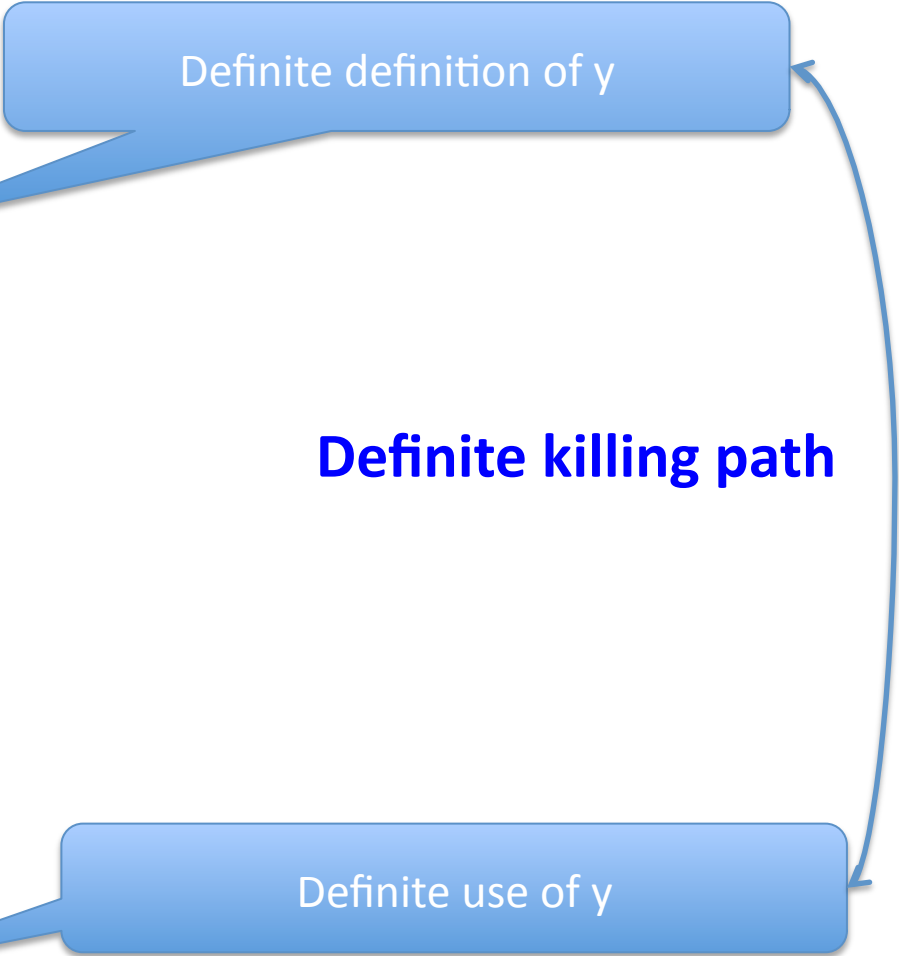
Definite use of y

```
proc(int cond1, int cond2) {  
    int x, y, z, *p;  
    z = 17;  
    x = 13;  
    if (cond1) {  
        p = &y;  
        *p = z;  
    }  
    else {  
        if (cond2)  
            p = &x;  
        else  
            p = &z;  
        *p = 7 + z;  
        y = 53;  
        p = &x;  
    }  
    x = x + y + z;  
    *p = *p + 5;  
    y = x + y;  
    ...  
}
```

Definite definition of y

Definite killing path

Definite use of y

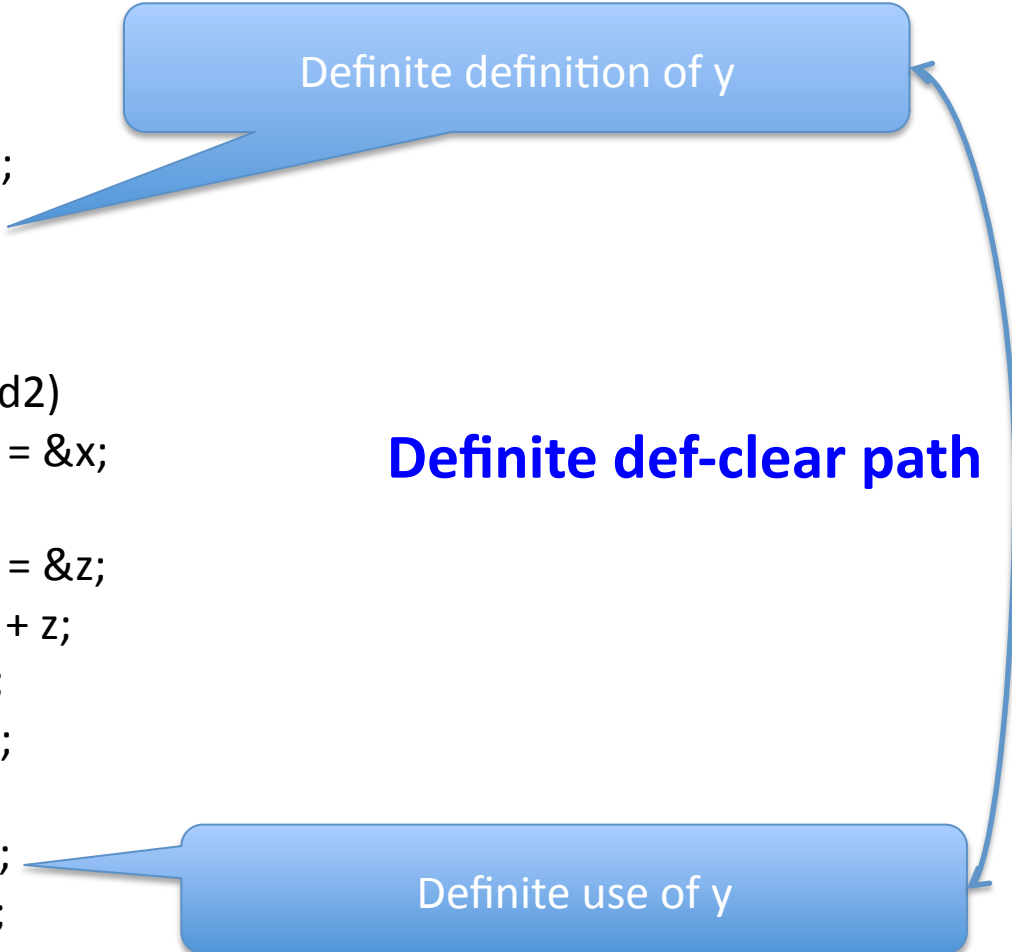


```
proc(int cond1, int cond2) {  
    int x, y, z, *p;  
    z = 17;  
    x = 13;  
    if (cond1) {  
        p = &y;  
        *p = z;  
    }  
    else {  
        if (cond2)  
            p = &x;  
        else  
            p = &z;  
        *p = 7 + z;  
        y = 53;  
        p = &x;  
    }  
    x = x + y + z;  
    *p = *p + 5;  
    y = x + y;  
    ...  
}
```

Definite definition of y

Definite def-clear path

Definite use of y



```
proc(int cond1, int cond2) {  
  int x, y, z, *p;  
  z = 17;  
  x = 13;  
  if (cond1) {  
    p = &y;  
    *p = z;  
  }  
  else {  
    if (cond2)  
      p = &x;  
    else  
      p = &z;  
    *p = 7 + z;  
    y = 53;  
    p = &x;  
  }  
  x = x + y + z;  
  *p = *p + 5;  
  y = x + y;  
  ...  
}
```

Definite definition of z

**Possible def-clear path
(through the else of cond1)**

Definite use of z

Classification of data dependencies that depend on pointer accesses (ii/ii)

Data dep.	Classification	Type of def and use	Characterization
Def-use pairs	Strong DU-pair	Both definite	Some definite def-clear paths No possible def-clear paths
	Firm DU-pair	Both definite	Some definite def-clear Some possible def-clear paths
	Weak DU-pair	Both definite	No definite clear-path Some possible def-clear paths
	Very weak DU-pair	Some possible	No definite clear-path Some possible def-clear paths

[Ostrand, Weyuker – TAV 1991]

[Orso, Sinha, Harrold – TOSEM 2004]

Can provide guidance to prioritize data dependencies
E.g., for test generation, according to the expected ease of executing them

Inter-procedural data flow analysis of object-oriented programs

- Context sensitivity requires **object sensitivity**
 - which object is the target of the method call?

Which m1,m2 does this program call?

```
aProgram(A obj){  
    A var =  
        Registry.getName("...");  
    var.m1(obj); //?????  
}
```

```
class A {  
    void m1(A a){  
        ...; a.m2();  
    }  
    void m2(){...}  
}
```

Which m1,m2 does this program call?

```
aProgram(A obj){  
    A var =  
        Registry.getByName("...");  
    var.m1(obj); //?????  
}
```

```
class A {  
    void m1(A a){  
        ...; a.m2();  
    }  
    void m2(){...}  
}
```

```
class B extends A {  
    void m1(A a){  
        ...; a.m2();  
    }  
    void m2(){...}  
}
```

It depends on the dynamic types of the variable <code>var</code> and the parameter <code>obj</code>		var	
		A	B
obj	A	A.m1, A.m2()	B.m1, A.m2()
	B	A.m1, B.m2()	B.m1, B.m2()

Inter-procedural data flow analysis of object-oriented programs

- Context sensitivity requires **object sensitivity**
 - Which object (class) is the receiver of the method call?
- In presence of dynamically dispatched messages
 - Multiple callees per call site are possible
 - The **receiver class set at each call site** is difficult to compute precisely
(necessitating itself inter-procedural data/control flow analysis)

Note the circular dependencies among the i) the inter-procedural data-flow analysis, ii) the program call graph and iii) receiver class sets ... → ... i)
Analysis needs to start from some initial “independent” approximation

Approaches to construct the call-graph

1. Pessimistic approach: Assume that all statically type-correct receiver classes are possible at every call site
 - Can be slightly improved by using use some form of simplified analysis
 - flow insensitive may-alias analysis
 - Intra-procedural data and control flow analysis
2. Refine iteratively the call graph while doing interprocedural analysis
 - Start with a call graph that contains only “main”
 - Assume all sets of receiver classes are initially empty

The pessimistic approach often leads to too coarse approximations

[Grove, Chambers. TOPLAS 2001]

Call graph – design dimensions

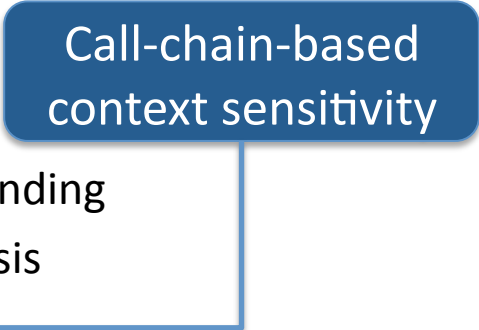
Terminology note

Call graph nodes are called “Contours”

Contour = Analysis time representation of a procedure

- Context-sensitive call graphs
 - Arbitrary num of contours (analysis-time views) of a procedure
 - The call graph edges represent, for each call site, the possible contours to analyze the callee
- To support the analysis, a procedure contour can record (some of)
 - Class sets for formal parameters, local variables, return value
 - The creation site of each instance
 - The class sets of the values within the propagating instances
- The design of a specific algorithm includes
 - The amount of information recorded in a contour (that must primarily allow to recognize the possible callees at a call site)
 - How to select the contour to analyze each of the callees at a call site

Some call graph construction Algorithms

- 0-CFA: context insensitive algorithms
 - A single contour to analyze each callee
 - There can still be some support to account for dynamic binding
 - Through interprocedural data flow and control flow analysis
- L-K-CFA family 
 - K: Consider K enclosing calling contours at each call site to select the target contour to analyze the callee
 - L: denotes the degree of context sensitivity with respect to (the contents of) different instance variables (according to the respective instantiation sites)
 - 0-1-CFA: a single separate contour for each call-site.
 - Adaptive algorithms use different levels of K/L in different call graph regions
 - Unbounded (but finite): a new contour for each non-recursive call
- CPA: Cartesian Product Algorithm
 - A contour for each element of the CP of the contours of the actual parameters
 - Thus, callers (that are similar enough) can share the contour

OO data flow testing

- At the method level, testing will target executions of method M, according to either/both
 - Intra-method def-use pairs
both def and use are in M (* ...and there is a def-clear path...)
 - Inter-method def-use pairs
def and use are in different methods – either M or some method called by M (*)
- At the class level, testing will target executions of sequences of method calls to class C
 - Intra-class def-use pairs
def and use are in different methods of C called independently (*)

References

- T Reps, S Horwitz, M Sagiv. Precise interprocedural dataflow analysis via graph reachability. POPL 1995
- MJ Harrold, ML Soffa. Efficient computation of interprocedural definition-use chains. ACM TOPLAS 1994
- TJ Ostrand, EJ Weyuker. Data flow-based test adequacy analysis for languages with pointers. TAV 1991
- A Orso, S Sinha, and MJ Harrold. Classifying data dependences in the presence of pointers for program comprehension, testing, and debugging. ACM TOSEM 2004
- D Grove, C Chambers. A framework for call graph construction algorithms. ACM TOPLAS 2001
- MJ Harrold, G Rothermel. Performing data flow testing on classes. SIGSOFT FSE 1994

Work @UniMiB/LTA

- G. Denaro, A. Gorla, M. Pezzè.
Contextual integration testing of classes. FASE 2008
- G. Denaro, A. Gorla, M. Pezzè.
DaTeC: contextual data flow testing of classes. ICSE Demo 2009
- G. Denaro, M. Pezzè, M. Vivanti.
On the right objectives of data flow testing. ICST 2014
- G. Denaro, M. Margara, M. Pezzè, M. Vivanti.
Dynamic data flow testing of object oriented systems. ICSE 2015

Do not hesitate to contact me if you would like to
check out the prototypes referred in the papers
denaro@disco.unimib.it