Interprocedural Analysis, Context Sensitivity in Practice

Announcements

- Quiz 1-2, HW1 now in Rainbow Grades
  - If you got points off, come by to get your paper
- HW2 due today?
- HW3: XTA Analysis
  - Will post the official assignment today
  - Submitty HW3 open tonight

Outline of Today's Class

- Interprocedural analysis: classical results
- Context-sensitive analysis in practice
  - Call-string-based context sensitivity
  - Cloning-based context sensitivity
  - Summary-based context sensitivity
- (Next time) The IFDS framework
  - Efficient and precise summary-based analysis
  - CFL-reachability

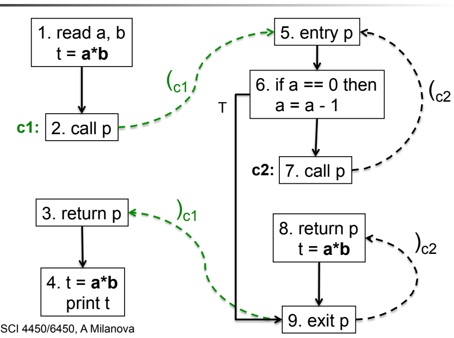
Reading

- Sharir and Pnueli's "Two approaches to Interprocedural dataflow analysis", 1981
- Dragon book, Chapter 12.1-3 Dragon book
- Thomas Reps, Susan Horwitz and Mooly Sagiv, "Precise, Interprocedural Dataflow Analysis via Graph Reachability, POPL'95

Classic Results and Ideas

- Sharir and Pnueli's "Two approaches to Interprocedural dataflow analysis", 1981
  - Amir Pnueli, Turing Award in 1996 for "For seminal work introducing temporal logic into computing science and for outstanding contributions to program and system verification."
- A finite lattice of dataflow facts
- Distributive transfer functions
- No local variables, and no parameter passing

Sharir and Pnueli Example (Available Expressions)



Sharir and Pnueli Example

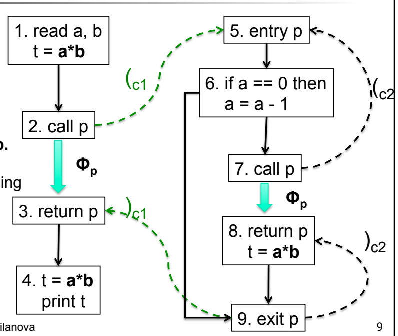
- Expression **a*b** is NOT available at 4 if we consider _all_ paths
  - E.g., along 1,2,5,6,**7,5**,6,**9,3**,4 **a*b** gets "killed" due to **a = a – 1**, and it is not recomputed
- Expression **a*b** is available at 4 if we consider only realizable paths
  - Path 1,2,5,6,7,5,6,9,3,4 is unrealizable because return edge 9,3 does not match the call edge 7,5
    - 1,2,5,6,**7,5** … **9,8**,9 ...
    - We know "kill" 6,7 is succeeded by 7,5, which must be balanced with 9,8, which is succeeded by "gen"

Functional Approach to Interprocedural Dataflow Analysis

- Operates on "intraprocedural" property space
- Computes summary transfer functions Φ_p that summarize effect of procedure **p**
- Reduces problem to intraprocedural case:
  - in(**return p**) = Φ_p(in(**call p**))
  - thus, avoids propagation from callee along the **exit p - - -> return p** edge!
- S_{FA}**(j)** is the solution at **j** computed by functional approach

Functional Approach

Phase 1:
Compute a **summary transfer function** Φ_p that captures effect of **p.**
(Assume our Φ_p is the **identity function**: nothing gets generated and nothing gets killed.)



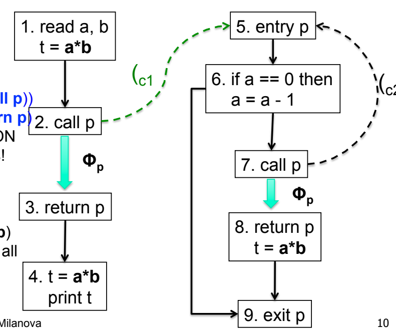
Functional Approach

Phase 2:
Dataflow analysis:
- At **return p**

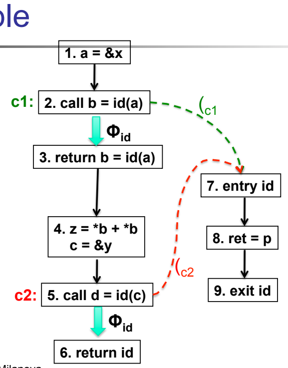
in(**return p**) = Φ_p(in(**call p**))
out(**return p**) = in(**return p**)
AVOIDS PROPAGATION along exit-return edges!

- At **entry p**

in(**entry p**) = V in(**call p**)
(propagates facts from all callers to callee)



Functional Approach: Another Example

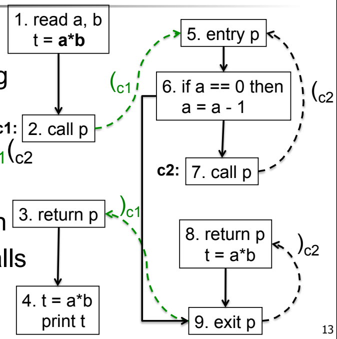


Computing Summary Transfer Functions

- For certain lattices and function spaces, we can compute summary transfer functions
  - The IFDS framework we discuss today
- In general, not clear how to compute Φ's efficiently
- Ad-hoc approaches/approximation when computing Φ's for specific monotone function spaces (points-to analysis, taint analysis)

Call String Approach to Interprocedural Dataflow Analysis

- A call string records outstanding calls in path
- E.g., call string $(_{c1}(_{c2}$ denotes that "we got there" on a path with outstanding calls at **c1** and at **c2**



Call String Approach

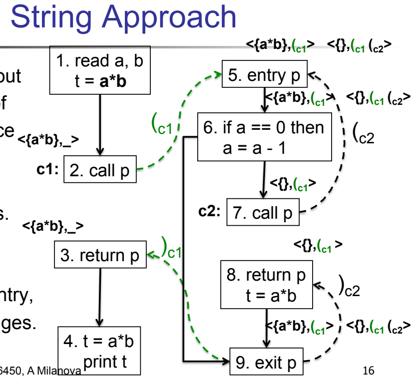
- Tags solutions per program point with corresponding call string
- Multiple tagged solutions per program point **j** in **p**:
  - Sharir and Pnueli Example:
  - We have **< { a*b }, $(_{c1}$ >**, **< { }, $(_{c1}(_{c2}$ >** at **6**
  - Meaning: **a*b** is available at **6** on paths with outstanding call string c1, but it is not available on paths with outstanding call string c1 c2

Call String Approach

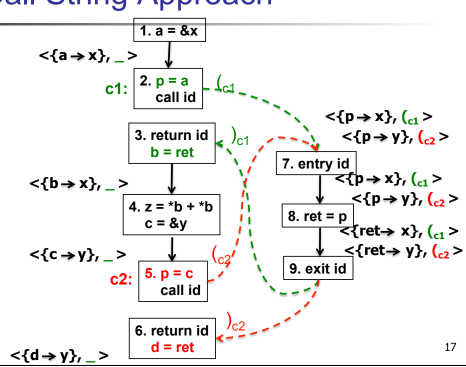
- Apply original transfer functions point-wise
- Apply on elements of the original, i.e., "intraprocedural" dataflow lattice
  - { **a*b** }, { **a*b**, **a+b** }, {}, etc.

Call String Approach

1. Extend in/out sets to sets of "tagged" lattice elements.
2. Apply orig. transfer funcs. point-wise.
3. Extend to handle call-entry, exit-return edges.



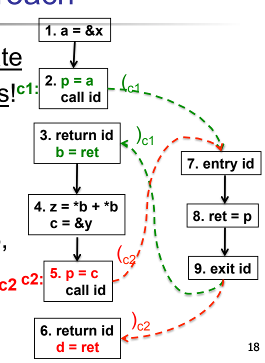
Call String Approach



Call String Approach

- At exit nodes, propagate only matching call strings!

<{ret →x}, $(_{c1}$ >,
< {ret→ y}, $(_{c2}$ > at **9**
Propagate **{ret→y}** to **6**, thus, **{d →y}**, because $)_{c2}$ matches call string $(_{c2}$

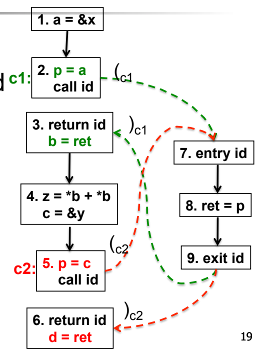


Call String Approach

- What is $S_{CS}(8)$?

Union of **<p ⟶x, $($**$_{c1}$ **>** and **<p⟶ y, $($**$_{c2}$ **>** so $S_{CS}(8)$ is graph { **p ⟶x, p ⟶y** }

- What is $S_{CS}(4)$?
- What is $S_{CS}(6)$? (out(6) more precisely)



Sharir and Pnueli, Key Result

- $\mathbf{S_{FA}(j)}$ is the solution at **j** computed by the functional approach
- $\mathbf{S_{CS}(j)}$ is the solution at **j** computed by the call string approach
- For (certain) distributive functions and finite lattices

$$\mathbf{S_{FA}(j) = S_{CS}(j) = MORP(j)}$$

- Caveats?

Sharir and Pnueli, Key Result

- Caveats
  - Summary functions difficult to compute
  - With recursion, infinite call strings, $\mathbf{S_{CS}}$ is infinite
  - Even for distributive functions and finite lattices, $\mathbf{S_{FA}}$ and $\mathbf{S_{CS}}$ cannot be computed (efficiently)

Outline of Today's Class

- Interprocedural analysis: classical results
- Context-sensitive analysis in practice
  - Call-string-based context sensitivity
  - Cloning-based context sensitivity
  - Summary-based context sensitivity
- (Next time) The IFDS framework
  - Efficient and precise summary-based analysis
  - CFL-reachability

Context-Sensitive Analysis In Practice

- Transfer functions are not distributive
- Local variables, flow of values from actual arguments to formal parameters, and from return to left-hand-side
- Procedures have side effects!
- Sometimes there is no call graph!
  - Function pointers, virtual calls, functions as first-class values
- Parameter passing mechanisms

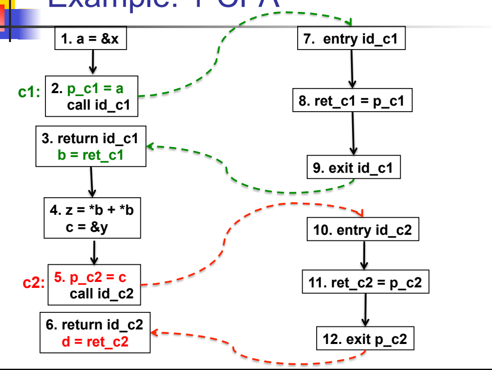
Context-Sensitive Analysis In Practice

- Context-sensitive analysis in practice: ad-hoc variants of Sharir and Pnueli's call string and functional approaches
- Call string approach
  - More intuitive than functional approach
  - Virtually universally applicable, widely used
- Functional approach
  - Better approach, whenever applicable
  - More difficult to implement
  - Better precision and better scalability, in general

Call String-Based Context Sensitivity

- Calling context is defined as the content of the entire stack
- Call-string-based context-sensitivity uses a _static_ call string as abstraction of the stack
- k-CFA: distinguishes context by **k** most recent call sites that lead to **p**
  - make a "copy" of procedure **p** for each _static_ call string of length **k**
- 1-CFA: "inline" **p** at each call site of **p**

Example: 1-CFA



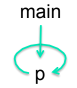
Problems?

```
   main:              id:
    …
    a = &x;           int* id(int* p) {
c1: b = id(a);          c3: return id_impl(p);
    z = *b + *b;      }
    c = &y;           …
c2: d = id(c);        int* id_impl(int* p) {
    …                   return p;
                      }
```

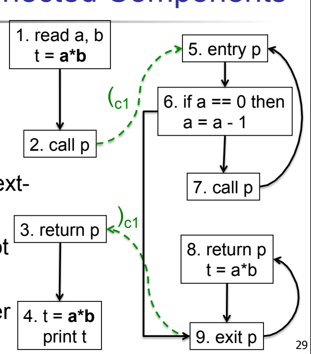

Problems?

- Exponential growth
  - Efficient data structures (Binary Decision Diagrams) can make full call string practical
- 2-CFA and 3-CFA are popular string lengths
- Recursion renders infinite call strings
- One (common) approach with recursion
  - Collapse strongly connected components in call graph into one big blob. Analyze blob as single procedure, context-insensitively
  - Analyze acyclic call graph with full call string

Strongly-Connected Components



- **p** forms a SCC. Treat calls within **p** context-insensitively (thus concluding that **a*b** is not available at 4 ☹)
- Analyze **p** per c1, other calling contexts if any



Recall: Points-to Analysis for Java (PTA)

- Saw in context of class analysis framework
- Context-insensitive, flow-insensitive analysis
- Syntax

Object allocation:	a_i: **x = new A // o_i**
Assignment:	**x = y**
Field Write:	**x.f = y**
Field Read:	**x = y.f**
Virtual call:	c_i: **x = y.m(z)**

Recall: PTA

- Next, define the analysis semantics
- Transfer functions (constraints) over syntax
  - E.g., Allocation **x = new A // o_i**

    for each reachable method **m**

    for each Allocation site **x = new A // o_i** in **m**

    { o_i } $\sqsubseteq$ pts(x)
  - Note: **pts(x)** denotes the points-to set of **x**
- Natural progression: RTA => XTA => 0-CFA => PTA

Recall: PTA
Transfer Functions (Constraints)

a_i: x = new A // o_i **{ o_i } $\sqsubseteq$ pts(x)**

x = y **pts(y) $\sqsubseteq$ pts(x)**

x.f = y for each **o** in **pts(x). pts(y) $\sqsubseteq$ pts(o.f)**

x = y.f for each **o** in **pts(y). pts(o.f) $\sqsubseteq$ pts(x)**

c_i: x = y.m(z)

for each **o** in **pts(y)**

let **m'(this,p,ret) = resolve(o,m)** in

{ o } $\sqsubseteq$ pts(this)

pts(z) $\sqsubseteq$ pts(p) **pts(ret) $\sqsubseteq$ pts(x)**

PTA Example

A a = new A(); // o_1

X x = new X(); // o_2

c1: a.set(x);

A a2 = new B(); // o_3

X x2 = new Y(); // o_4

c2: a2.set(x2);

// set(X p) { this.f = p; }

Boolean Expression Hierarchy: PTA

```
main() {
  Context theContext = new Context();

  BoolExp or1 = new OrExp(new VarExp("X"),  // or1
                          new VarExp("Y"));
  BoolExp or2 = new OrExp(new Constant(true), // or2
                          new Constant(false));

  boolean result1 = or1.evaluate(theContext);
  boolean result2 = or2.evaluate(theContext);
}
```


Boolean Expression Hierarchy: PTA

```
public class OrExp extends BoolExp {
  private BoolExp left; private BoolExp right;

  public OrExp(BoolExp left, BoolExp right) {
    this.left = left;
    this.right = right;
  }

  public boolean evaluate(Context c) {
    private BoolExp l = this.left;
    private BoolExp r = this.right;
    return l.evaluate(c) || r.evaluate(c);
  }
}
```

pts(this) = { or_1, or_2 }
pts(left) = { var_1, c_1 }, pts(right) = { var_2, c_2 }
pts(or_1.left) = pts(or_2.left) = { var_1, c_1 } !!!

pts(l) = { var_1, c_1 }
pts(r) = { var_2, c_2 }

Boolean Expression Hierarchy: How About 1-CFA?

```
main() {
  Context theContext = new Context();

  c1: BoolExp or1 = new OrExp(new VarExp("X"),  // or1
                              new VarExp("Y"));
  c2: BoolExp or2 = new OrExp(new Constant(true), // or2
                              new Constant(false));

  c3: boolean result1 = or1.evaluate(theContext);
  c4: boolean result2 = or2.evaluate(theContext);
}
```

What If We Changed Boolean Expression Hierarchy? 1-CFA?

```
public abstract class BinaryExp extends BoolExp {
  private BoolExp left; private BoolExp right;

  public BinaryExp(BoolExp left, BoolExp right) {
    this.left  = left; this.right = right;
  }
  …
}
public class OrExp extends BinaryExp {
  public OrExp(BoolExp left, BoolExp right) {
    c5: super(left,right); // call to constructor BinaryExp.<init>
  }
  …
}
```


What If We Changed Boolean Expression Hierarchy: 1-CFA?

```
main() {
  Context theContext = new Context();

  c1: BoolExp or1 = new OrExp(new VarExp("X"),  // or1
                              new VarExp("Y"));
  c2: BoolExp or2 = new OrExp(new Constant(true), // or2
                              new Constant(false));

  c3: boolean result1 = or1.evaluate(theContext);
  c4: boolean result2 = or2.evaluate(theContext);
}
```


Cloning-based Context Sensitivity

- Remember, calling context is the content of the entire call stack
- Cloning-based context sensitivity uses program state of interest as abstraction of the stack
- Clone (i.e., copy) a procedure for each program state of interest, i.e., "calling context"
- A hybrid of functional and call-string

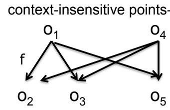
Cloning-Based Context Sensitivity

A a = new A(); // o_1
c1: a.set(new X()); // o_2
c2: a.set(new X()); // o_3

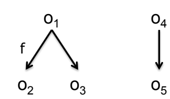
A a2 = new B(); // o_4
c3: a2.set(new Y()); // o_5

// set(X p) { this.f = p; }

Flow-insensitive, context-insensitive points-to graph



Flow-insensitive 1-CFA context-sensitive points-to graph (also "ground truth")



Cloning-Based Context Sensitivity

- It is more effective if we "cloned" method **set** per receiver object rather than per call site

A a = new A(); // o_1
c1: a.set_o_1(new X()); // o_2
c2: a.set_o_1(new X()); // o_3
A a2 = new A(); // o_4
c3: a2.set_o_4(new Y()); // o_5

- Again, flow-insensitive and context-sensitive, reaches our "ground truth"

Cloning-Based Context Sensitivity

class A { <init>(X p) { this.f = p; } … }
class B extends A { <init>(X p) { c1: super(p); } }
Note: **super** calls **A.<init>(p)**
class C extends B { <init>(X p) { c2: super(p); } }

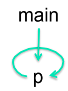
c = new C; // o_1
c3: c.<init>(new X()); // o_2
c2 = new C; // o_3
c4: c2.<init>(new X()); // o_4

1-CFA?
2-CFA?
3-CFA?

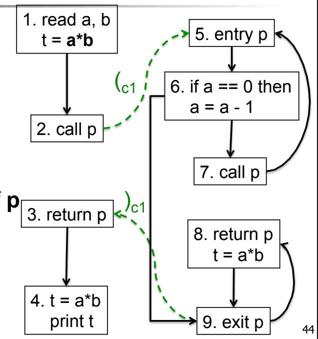
Summary-based Context Sensitivity

- Compute summary transfer functions
  - **x = id(y)** applies "add **x**⟶**a** for each **y**⟶**a**" (points-to for C example)
  - **p()** applies the "identity function" (Sharir and Pnueli's Available expressions example)
  - **a.set(x)** "sets field **f** of all objects **a** points to to point to the objects **x** points to" (PTA example)
- Phase 1: compute summary transfer functions
  - Collapse into SCC on call graph, then compute summaries bottom up
- Phase 2: propagate values into callees

Strongly-Connected Components



- **p** forms a SCC.
- Compute summary of **p** treating SCC as single procedure
- Summary of **p** says **a*b** is NOT available ☹



Summary-based Context Sensitivity

- For a class of lattices and transfer functions one can represent functions, and compute summary transfer functions efficiently!