Interprocedural Analysis and Context Sensitivity

Announcements

- HW2?
- Submitty page still not open… Will test toy programs

Outline of Today's Class

- Overview of homework
- Catch-up: Class analysis
- XTA
- 0-CFA
- Points-to Analysis (PTA)

XTA

R is the set of reachable methods
S_m is the set of types that flow to method **m**
S_f is the set of types that flow to field **f**

1. { main } $\sqsubseteq$ **R**

2. for each method **m** $\in$ **R** and
   each new site **new C** in **m**
   { **C** } $\sqsubseteq$ S_m

XTA

3. for each method **m** $\in$ **R**,
each virtual call **y.n(z)** in **m**,
each class **C** in **SubTypes(StaticType(y))**$\cap S_m$
and **n'**, where **n'** = **resolve(C,n)**
    { **n'** } $\sqsubseteq$ **R** // add **n'** to **R** if not already there
    { **C** } $\sqsubseteq$ $S_{n'}$ // add **C** to $S_{n'}$ if not already there
    S_m $\cap$ **SubTypes(StaticType(p))** $\sqsubseteq$ $S_{n'}$
    $S_{n'}$ $\cap$ **SubTypes(StaticType(ret))** $\sqsubseteq$ S_m
(**p** denotes the parameter of **n'**, and **ret** denotes the return of **n'**)

XTA

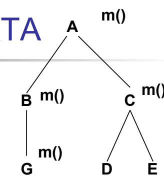
4. for each method **m** $\in$ **R**,
each field read **x = y.f** in **m**
    $S_f \sqsubseteq S_m$

5. for each method **m** $\in$ **R**,
each field write **x.f = y** in **m**
    $S_m \cap$ **SubTypes(StaticType(f))** $\sqsubseteq S_f$

Practical Concerns

- Multiple parameters
- Direct calls
  - either static invoke calls or
  - special invoke calls
- Array reads and writes!
- Static fields
- See Tip and Palsberg for more

Example: RTA vs. XTA




```
public class A {
  public static void main() {
    n1();
    n2();
  }
  static void n1() {
    A a1 = new B();  // S_n1 approximates locals in n1
    a1.m();          // S_n1 = { B }
  }
  static void n2() {
    A a2 = new C();  // S_n2 approximates locals in n2
    a2.m();          // S_n2 = { C }
  }
}
```


Boolean Expression Hierarchy: RTA vs. XTA

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

  public AndExp(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);
  }
}
```


Boolean Expression Hierarchy: RTA vs. XTA

```
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);
  }
}
```


Boolean Expression Hierarchy: RTA vs. XTA

```
main() {
  Context theContext = new Context();
  BoolExp x = new VarExp("X");
  BoolExp y = new VarExp("Y");
  BoolExp exp = new AndExp(
                  new Constant(true), new OrExp(x, y) );
  theContext.assign(x, true);
  theContext.assign(y, false);
  boolean result = exp.evaluate(theContext);
}
```


0-CFA

- 0-CFA stands for 0-level Control Flow Analysis, where "0-level" stands for context-insensitive analysis
  - Will see 1-CFA, 2-CFA, … k-CFA
- Improves on XTA by storing even more information about flow of class types

0-CFA

R is the set of reachable methods

S_v is the set of types that flow to variable **v**

S_f is the set of types that flow to field **f**

1. { main } $\sqsubseteq$ **R**

2. for each method **m** $\in$ **R** and
   each new site **x = new C** in **m**
   { C } $\sqsubseteq S_x$

0-CFA

3. for each method **m** $\in$ **R**,
each virtual call **x = y.n(z)** in **m**,
each class **C** in S_y
and **n'**, where **n' = resolve(C,n)**

{ n' } $\sqsubseteq$ R

{ C } $\sqsubseteq S_{this}$

$S_z \cap$ **SubTypes(StaticType(p))** $\sqsubseteq S_p$

$S_{ret} \cap$ **SubTypes(StaticType(x))** $\sqsubseteq S_x$

(**this** is the implicit parameter of **n'**, **p** is the parameter of **n'**, and **ret** is the return of **n'**)

0-CFA

4. for each method **m** $\in$ **R**,
each field read **x = y.f** in **m**

$S_f \cap$ **SubTypes(StaticType(x))** $\sqsubseteq S_x$

5. for each method **m** $\in$ **R**,
each field write **x.f = y** in **m**

$S_y \cap$ **SubTypes(StaticType(f))** $\sqsubseteq S_f$

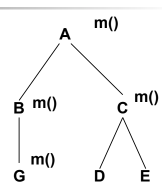
0-CFA

6. for each method **m** $\in$ **R**,
each assign stmt **x = y** in **m**

$S_y \cap$ **SubTypes(StaticType(x))** $\sqsubseteq S_x$

Example: XTA vs. 0-CFA

```
public class A {
   public static void main() {
      A a1 = new B();
      a1.m();

      A a2 = new C();
      a2.m();
   }
}
```



Boolean Expression Hierarchy: XTA vs. 0-CFA

```
main() {
   Context theContext = new Context();
   BoolExp x = new VarExp("X");
   BoolExp y = new VarExp("Y");
   BoolExp exp = new AndExp(
                    new Constant(true), new OrExp(x, y) );
   theContext.assign(x, true);
   theContext.assign(y, false);
   boolean result = exp.evaluate(theContext);
}
```

Boolean Expression Hierarchy: XTA vs. 0-CFA

```
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);
  }
}
```

$S_{OrExp.left(field)}$ = { VarExp }
$S_{OrExp.right(field)}$ = { VarExp }

$S_{l(local)}$ = { VarExp }
$S_{r(local)}$ = { VarExp }

PTA

- Widely referred to as Andersen's points-to analysis for Java
- Improves on 0-CFA by storing information about **objects**, not classes
  - A a1 = new A(); // $\mathbf{o_1}$ ... $\mathbf{o_1}$**.f**
  - A a2 = new A(); // $\mathbf{o_2}$ ... $\mathbf{o_2}$**.f**

PTA

R is the set of reachable methods

pts(v) is the set of objects that **v** may point to

pts(o.f) is the set of objects that field **f** of object **o** may point to

1. **{** main **}** ⊑ **R**

2. for each method **m** ∈ **R** and
   each new site **i: x = new C** in **m**
   { o_i **}** ⊑ **pts(x) // instead of C, we have** o_i

PTA

class_of(o) returns the class of object **o**

3. for each method **m** ∈ **R**,
each virtual call **x = y.n(z)** in **m**,
each class $\mathbf{o_i}$ in **pts(y)**
and **n'**, where **n'** = **resolve(class_of(**o_i**),n)**
   { n' } ⊑ **R**
   { o_i **}** ⊑ **pts(this)**
   pts(z) ∩ **SubTypes(StaticType(p))** ⊑ **pts(p)**
   pts(ret) ∩ **SubTypes(StaticType(x))** ⊑ **pts(x)**
(**this** is the implicit parameter of **n'**, **p** is the parameter of **n'**, and **ret** is the return of **n'**)

PTA

4. for each method **m** ∈ **R**,
each field read **x = y.f** in **m**
   for each object **o** ∈ **pts(y)**
      pts(o.f) ∩ **SubTypes(StaticType(x))** ⊑ **pts(x)**

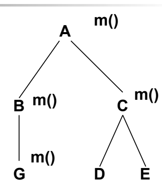
5. for each method **m** ∈ **R,**
each field write **x.f = y** in **m**
   for each object **o** ∈ **pts(x)**
      pts(y) ∩ **SubTypes(StaticType(f))** ⊑ **pts(o.f)**

PTA

6. for each method **m** ∈ **R**,
each assign stmt **x = y** in **m**
   pts(y) ∩ **SubTypes(StaticType(x))** ⊑ **pts(x)**

Example: 0-CFA vs. PTA

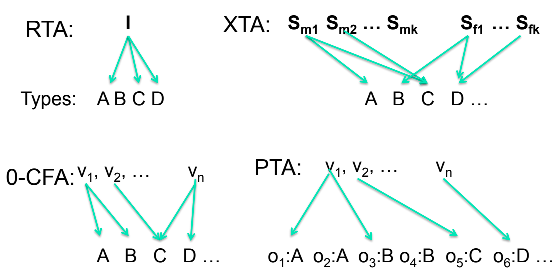
```
public class A {
   public static void main() {
      X x1 = new X();    // o1
      A a1 = new B();   // o2
      x1.f = a1;  // o1.f points to o2
      A a2 = x1.f;  // a2 points to o2
      a2.m();

      X x2 = new X();    // o3
      A a3 = new C();   // o4
      x2.f = a3;  // o3.f points to o4 equiv. pts(o3.f) = { o4 }
      A a4 = x2.f; // a4 points to o4
      a4.m();
   }
}
```



The Big Picture

- All fit into our monotone dataflow framework!
- Flow-insensitive, context-insensitive
  - Least solution of S = f_j(S) **V** S
- Algorithms differ mainly in "size" of S
  - RTA: only 2 kinds of statements; Lattice?
  - XTA: expands to all statements; Lattice?
  - 0-CFA: all statements; Lattice?
  - PTA (Points-to analysis): all statements; Lattice elements are points-to graphs

The Big Picture



Outline of Today's Class

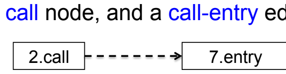
- Interprocedural Control Flow Graph (ICFG)
- Realizable paths, Meet-Over-all-Realizable-Paths (MORP) solution
  - Also denoted as MVP or MRP
- Classical results on interprocedural analysis
  - Call-string approach
  - Functional approach
- Context-sensitive analysis in practice

Outline of Today's Class

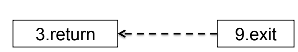
- Reading:
- Sharir and Pnueli, "Two approaches to interprocedural dataflow analysis", 1981
- Chapter 12.1-3 Dragon book

Interprocedural Control Flow Graph (ICFG)

- Add procedure entry node and exit node
- At each procedure call add
  - A call node, and a call-entry edge

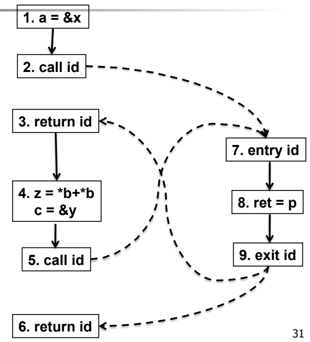


  - A return node, and an exit-return edge



Interprocedural Control Flow Graph (ICFG)

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

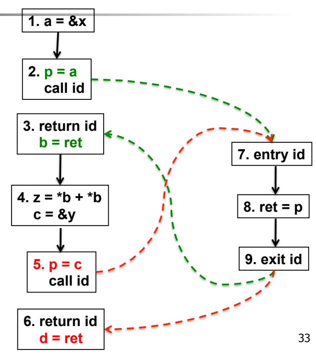


Context-Insensitive Analysis

- Add explicit assignments at call/return
  - E.g., **x = id(y)**
  - **p = y** models flow from actuals to formals
  - **x = ret** models flow from return to lhs
- Treat ICFG as one big CFG, and apply worklist algorithm
- Problem: merges data from different contexts
- Goal: track "realizable paths". Context-sensitive analysis tracks "realizable paths"

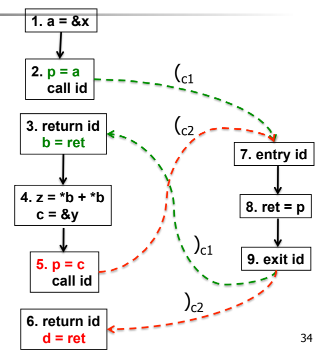
Infeasible Paths

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



Realizable Paths

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

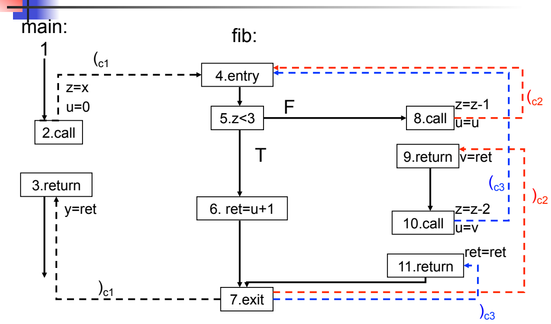


Another Example

```
int fib(int z, int u) {
   if (z<3) {
        return u+1;            /* ret = u+1;
   } else {                     auxiliary variable ret
        c2: v = fib(z-1,u);     holds the return values */
        c3: return fib(z-2,v)
   }
}
…
c1: y = fib(x,0);
…
```

What does **fib** compute? Here **z** and **u** are formal parameters; **ret** is the special variable holding the return value.

Another Example



Realizable Paths (RP)

- Context-free grammar!
- Same-level (balanced) path (**SLP**):

M ::= **e** **e** denotes intraprocedural edge

| $(_{ci}\ M\ )_{ci}$ captures path from call to return

| $M\ M$

  - An **intra**procedural edge is annotated with **e**
  - Call edge that originates at call site c_i is $(_{ci}$
  - Corresponding return edge is $)_{ci}$
- A path **p,** from **m** to **n,** is in $\mathbf{SLP}_{m,n}$ iff string along **p** is in language described by M

Realizable Paths (RP)

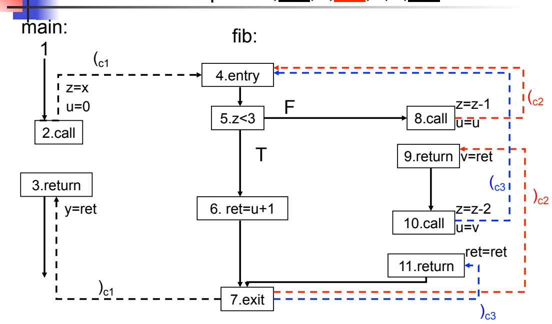
- Another grammar, describes paths with outstanding calls (i.e., calls not yet returned):

C ::= $(_{ci}$ | $M\ (_{ci}$ | $C\ M$

- A path from entry node 1 to node **n** is in $\mathbf{RP}_{1,n}$ iff the string from 1 to **n** is in the language generated by either M or C
  - E.g., in **fib**, 1,2,4,5,6,7 is in **RP** but 1,2,4,5,8,4,5,6,7,3 is not in **RP**

Is p1 = 1,2,4,5,6,7 in RP?

Is p2 = 1,2,4,5,8,4,5,6,7,3 in RP?



Meet Over All Realizable Paths (MORP)

- $\mathbf{MORP\ (n)} = \vee\ f_{n_k} \circ f_{n_{k-1}} \circ \ldots\ f_{n_2} \circ f_1(\text{init})$

  $p=(1,n_2\ldots n_k,n)$ is a path in $RP_{1,n}$

(° denotes function composition)

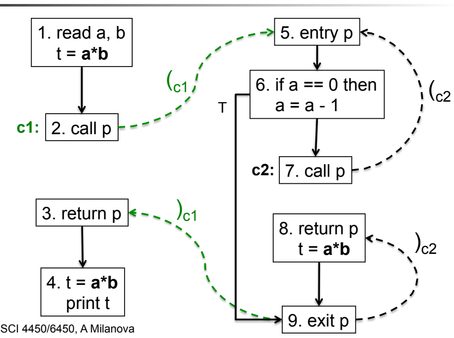
  - Also called MVP (meet over all **valid** paths) or just MRP

- MORP(n) ≤ MOP(n). Why?
- May be undecidable even for lattices of finite height
- Goal: encode context and restrict analysis over realizable paths, as much as possible

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

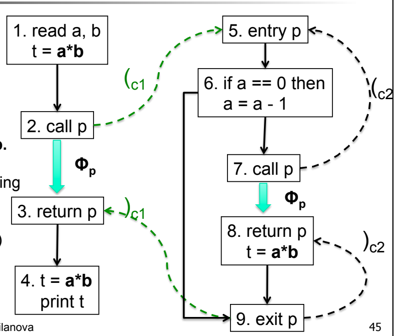
- Expression **a*b** is NOT available at 4 if we consider _all_ paths
  - E.g., along 1,2,5,6,7,5,6,9,3,4 **a*b** gets "killed" due to **a = a – 1**, and it is not recomputed
- Expression **a*b** is available at 4 if we consider only realizable paths
  - Path 1,2,5,6,7,5,6,9,3,4 is unrealizable because 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 unchanged property space
- Computes summary transfer functions Φ_p that summarize effect of procedurep **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 (simplifying things a bit)



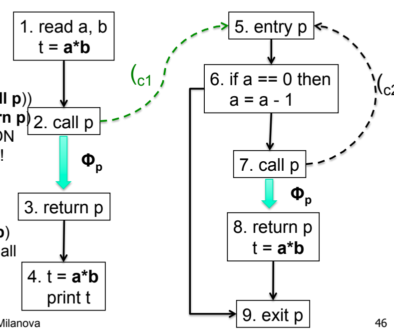
Functional Approach

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)

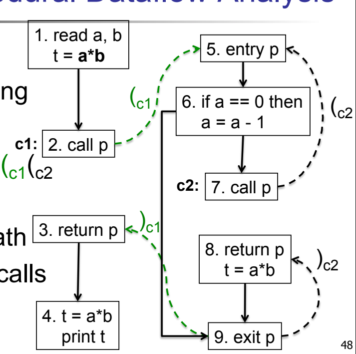


Computing Summary Transfer Functions

- 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

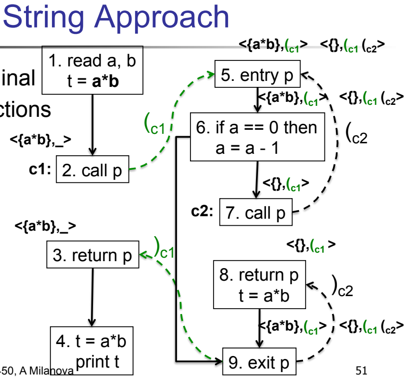
- Tags solutions per program point with corresponding call string
- Multiple tagged solutions per program point **j** in **p**:
  - Sharir and Pnueli Example:
  - We have **< { a*b }, $(_{c1}$ >**, **< { }, $(_{c1}(_{c2}$ >** at **6**
  - Meaning: **a*b** is available at **6** on paths with outstanding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

Call String Approach

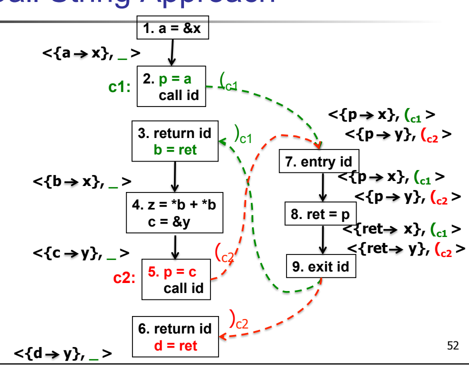
- Apply original transfer functions point-wise
- Apply on original dataflow lattice elements
  - { **a*b** }, { **a*b**, **a+b** }, {}, etc.

Call String Approach

- Apply original transfer functions point-wise



Call String Approach

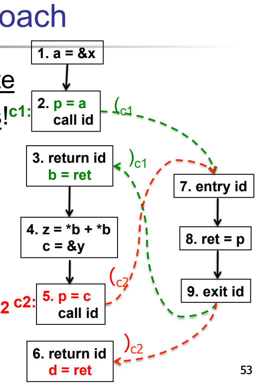


Call String Approach

- 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 $(_{c2}$

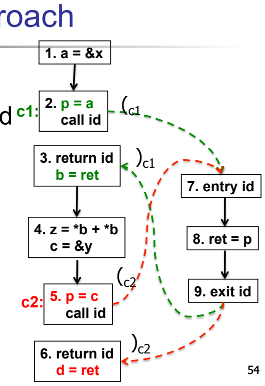


Call String Approach

- 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)