Learn to Build Automated Software Analysis Tools with Graph Paradigm and Interactive Visual Framework

GIAN Jaipur, September 12-16, 2016

Suresh C. Kothari
Richardson Professor
Department of Electrical and Computer Engineering

Ben Holland, Iowa State University

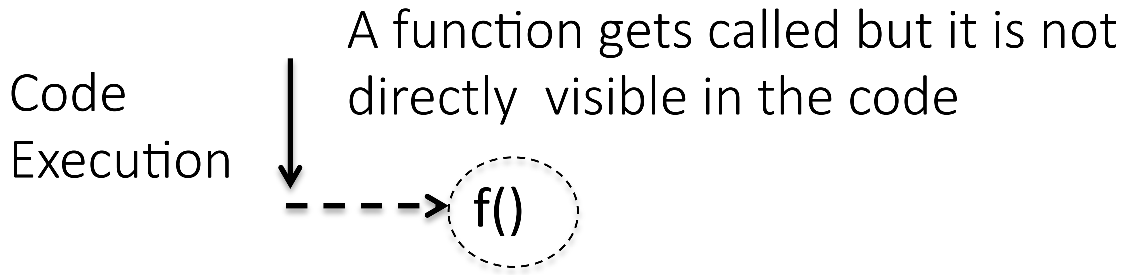
Module III: Two Central Program Analysis Problems

Acknowledgement: **Team members at Iowa State University and EnSoft, DARPA contracts FA8750-12-2-0126 & FA8750-15-2-0080**

Module Outline

- A *control flow problem* – where does the control go from a call site?
- Indirect control flow due to dynamic dispatch: (a) function pointers in procedural languages, (b) type hierarchy in object-oriented languages.
- A research survey of call graph construction algorithms
- A *data flow problem* – what are the objects a pointer points to?
- Two key points-to algorithms: (a) Streensgaard, (b)Anderson
- Reference implementations and smart views for teaching and research on program analysis algorithms.

Invisible Flow – two types

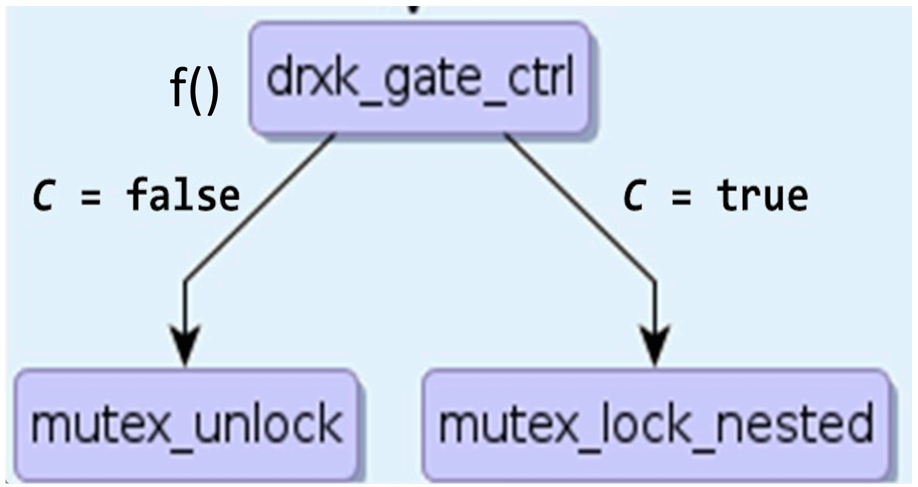

A function gets called but it is not directly visible in the code
Code Execution
f()

Two Types:

1. The call mechanism is visible (e.g. function pointer).
2. The call mechanism is not visible (e.g. a call due to an interrupt).

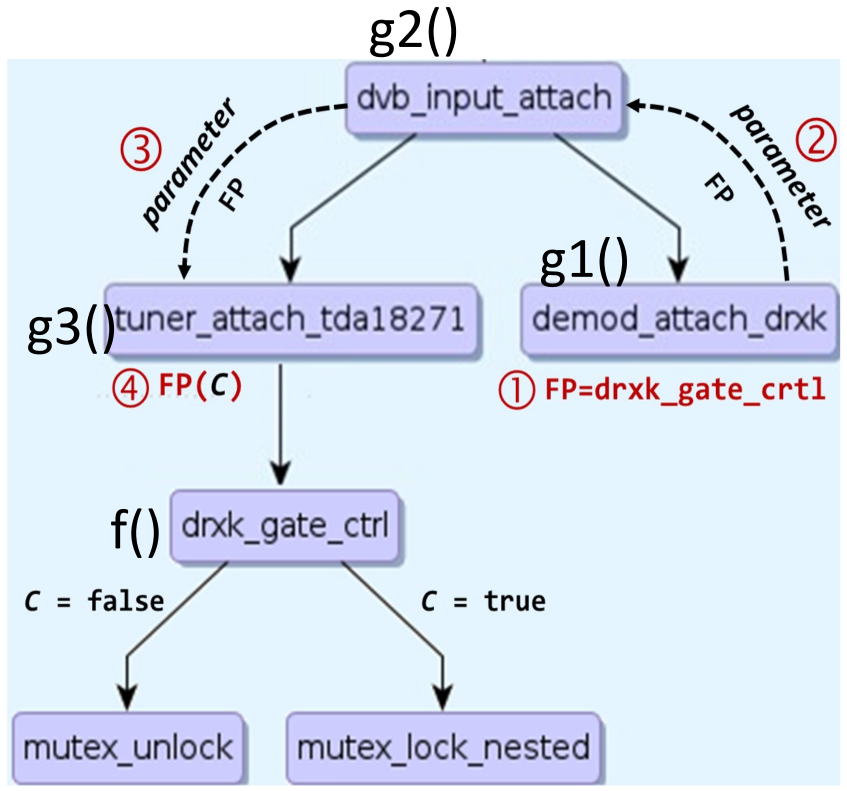
A motivating Linux example

f() not called in the code, is it dead code?



No, there is a call to f() using a function pointer.

A bug found by resolving invisible flows



g3() has two function pointer calls:

1. FP( ) with (C==T)
2. FP( ) with (C==F)

However, there is a RETURN in between the two calls. So, it is a bug.

1. g1() sets a function pointer FP to f()
2. g1() passes FP as a parameter to g2().
3. g1() passes FP as a parameter to g3().
4. g3() has two function pointer calls.

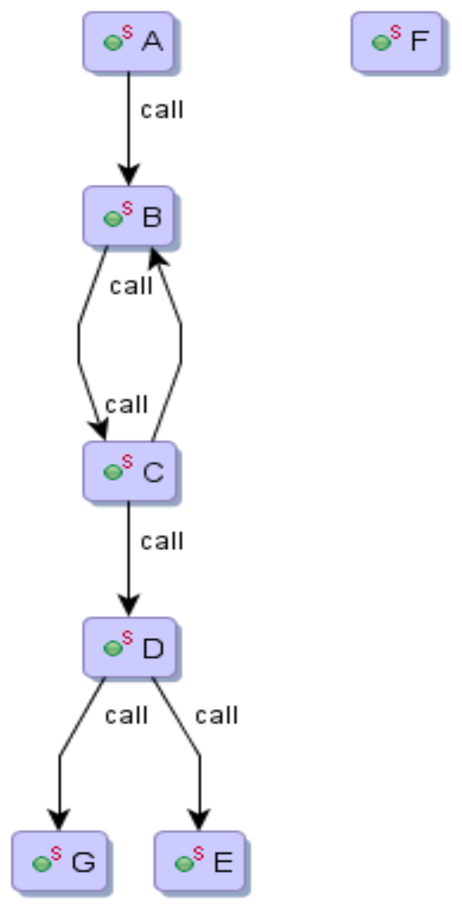
Research: call graph construction for object-oriented languages

- Problem: Develop algorithms to compute the set S_M(O.f()), the set of methods that may invoked for the call O.f().
- Goal: sound, precise, and scalable algorithms where
  - Sound implies the algorithm does not miss any methods – no false negatives
  - Precise implies the algorithm does not report any method that will never be called – no false positives.
- Smaller S_M implies better precision – how to achieve better precision without losing soundness?
- To improve precision, algorithms compute more information about O and/or f()

Major categories of CG algorithms

- Reachability Analysis (RA): Look for methods with the same name, the same parameter types/counts, and the return type.
- Class Hierarchy Analysis (CHA): RA + Use type hierarchy to constrain the set of possible methods.
- Rapid Type Analysis (RTA): RA + restrict to methods for the actually allocated types.
- Variable Type Analysis (VTA): RA + restrict the methods for the allocated objects to which P points to.

How do we produce a call graph?



Idea 1: Reachability Analysis (RA)

- For Every Method *m*
  - For Every Callsite
    - Add an edge (if one does not already exist) from *m* to any method with the same name as the method called in the callsite

- Refinement: Match methods with the same name AND the same return types and parameter types/counts

- Sound but not precise
  - We end up with a call graph that always has a call edge that *could* happen, but we have a lot of edges that also can not happen.
  - WHY? -> Static Dispatch vs. Dynamic Dispatch

Static Dispatch vs. Dynamic Dispatch

- Static Dispatch
  - Resolvable at compile time
  - Includes calls to Constructors and methods marked “static” (ex: main method)
  - Static methods do not require an object instance, they can be called directly
- Examples
  - Animal a = new Dog(); // static dispatch to new Dog()
  - Animal.runSimulation(a); // static dispatch to runSimulation()

Static Dispatch vs. Dynamic Dispatch

- Dynamic Dispatch
  - Resolvable at runtime
  - Includes calls to member methods (virtual methods)
  - Requires an object instance, they can be called directly
  - Very common in OO languages such as Java
- Examples
  - Animal a = new Dog(); // static dispatch to new Dog()
  - a.toString(); // dynamic dispatch to toString
  - a.getName(); // dynamic dispatch to dog's getName method

Terminology (slight digression)

```
Animal a = new Dog(); // static dispatch to new Dog()
a.toString(); // dynamic dispatch to toString
```

Callsite of String::toString() method (could be overridden in Dog, inherited from Animal or Object)

Receiver object

Declared type (Animal)

Runtime type (Dog)

Idea 2: Class Hierarchy Analysis (CHA)

- Compute the type hierarchy
- For each callsite, if the dispatch is static add the edge like normal, otherwise:
  - Perform RA with the added constraint that the target method must be in either
    - The direct lineage from Object to the receiving object's declared type (in the case that the target method is inherited)
    - The subtype hierarchy of the receiving object's declared type

- Sound and more precise than RA
  - We end up with a call graph that always has a call edge that *could* happen, and we have less edges that can not happen.
  - We can still do better…in terms of precision
  - Checkout the case of: Object o = …; o.toString();
    - We have to add an edge to every toString() method!

Idea 3: Rapid Type Analysis (RTA)

- Summary:
  - Look at the allocation types that were made
  - We can’t have a call to a method in a type that we never had an instance of (in the case of dynamic dispatches)
- Implementation (on-the-fly call graph construction)
  - Start with CHA
  - Worklist style algorithm starting at the main method
  - Remove a method from the worklist
    - Examine its new allocation types and the types of its parents
    - For each allocation type that matches a type in the CHA call graph, add a new edge
    - For each new edge add the called method to the worklist
- Described in detail in “Fast Static Analysis of C++ Virtual Function Calls – IBM”, 1996.
  - More precise than CHA in practice, but unsound!

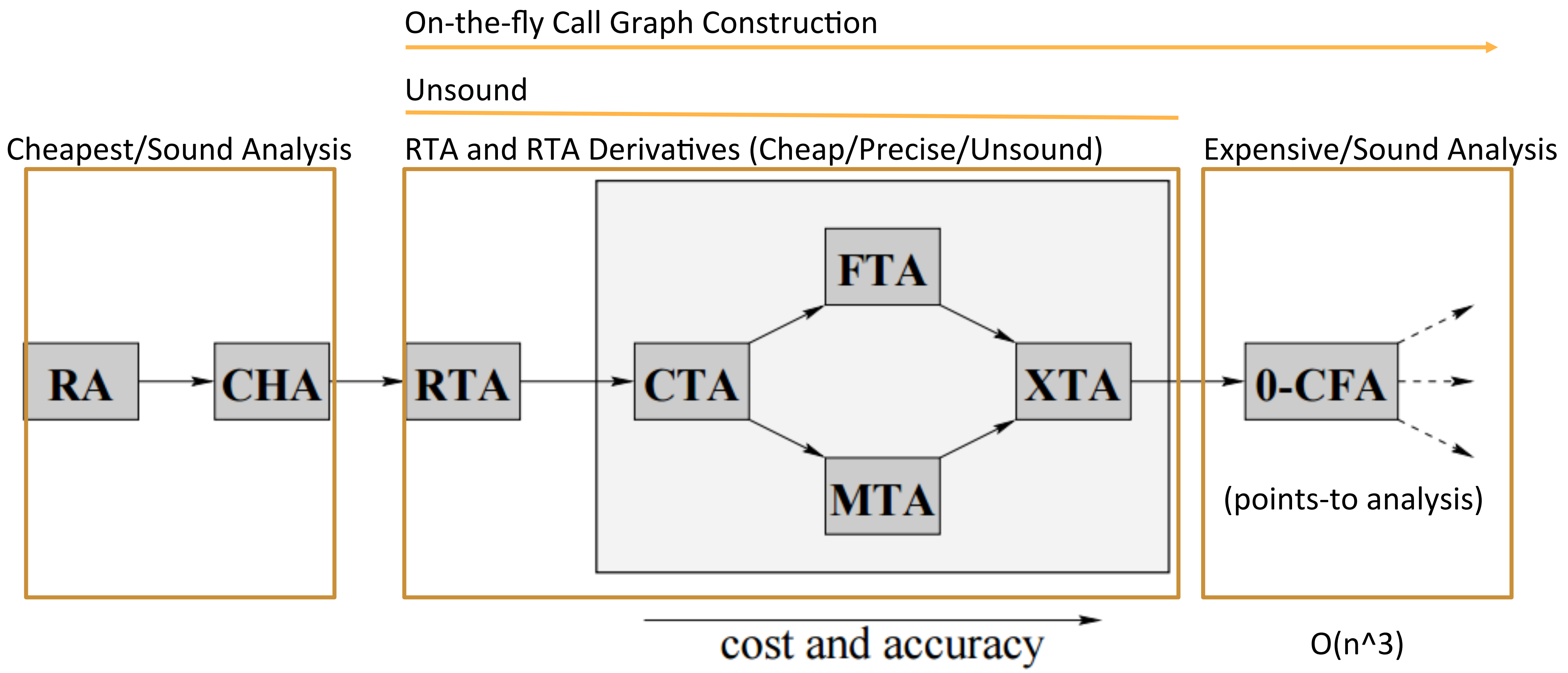
Idea 4: Rapid Type Analysis Improvements

- Method Type Analysis (MTA)
  - Idea: Restrict parent method types to types that could be passed through the method's parameters
  - Idea: Consider the statically typed return type of the dynamic dispatch callsite
- Field Type Analysis (FTA)
  - Idea: Consider that a method could write to a global variable, so any allocations reachable by a method are also reachable by a method that reads the same global variable
- Hybrid Type Analysis (XTA)
  - Combines MTA and FTA
  - Precision? More precise than RTA
  - Sound? No…Exceptions are not considered
- Paper: Scalable Propagation-Based Call Graph Construction Algorithms

Idea 5: Variable Type Analysis

- Idea: Track the allocation types to variables the callsites are made on.
  - This is a points-to analysis
- Implementation (one strategy)
  - For each new allocation, assign an ID to the new allocation site add each allocation site to the worklist
  - While the worklist is not empty
    - Remove an item from the worklist and propagate its point-to set (set of allocation ids) to every data flow successor (every assignment of the variable to another variable), adding each new variable to the worklist
    - If a callsite is made on the variable, look up the type of the allocation(s) and add a call edge (if parameters are reachable as a result, add the parameters to the worklist)

Call Graph Construction Algorithm Overview



Things to think about…

- Can we create a call graph of a library without an application?
  - How should we deal with "callback edges", that is the case when an application overrides a method in the library so that a dynamic dispatch callsite in the library actually goes to the application.
  - If we don't have all of the code we can't know what happens at runtime, but we can say something about what can't happen.

- How does Java Reflection affect call graph construction?
  - Java Reflection is a skeleton in the closest for most researchers, in most cases its not handled.

How to get the points-to graph at a program point?

- Note the two different questions:
  - Q1: Given a variable V, what is the points-to set P(V) ?
  - Q2: Given a variable V, what is the points-to set P(V,X) at a program point X?
- The Anderson’s algorithm answers Q1.
- Practical applications require the answer to Q2.

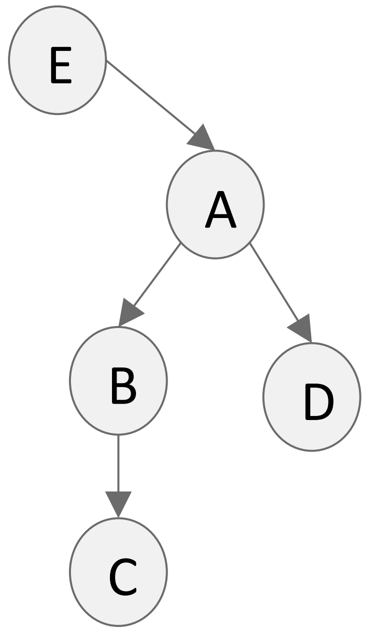
Anderson’s Algorithm

- Flow-Insensitive, Context-Insensitive algorithm
- Pointer Assignments as set constraints.
- v1,v2, v3, … vn denote pointers pointing to objects O1,O2,O3… On. P(vi) denotes points-to-set of vi.
- Rules
  - Initialize P(vi) = {Oi} ∀ 1 ≤ i ≤ n
  - If vi = vj; then P(vi) ⊇ P(vj)
  - If vi = vj and vj = vk; then P(vi) ⊇ P(vi) ∪ P(vj) ∪ P(vk)

Anderson’s Algorithm : Computational Complexity

- $O(n^3)$
- Explanation :
  - No. of assignments to consider is $O(n^2)$
  - Rule 2 requires iterating over all assignments for each pointer : $O(n)$
  - Hence complexity is $O(n^2*n) = O(n^3)$

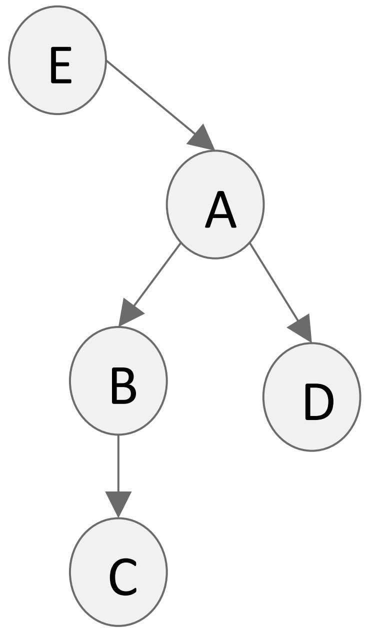
Computational Complexity (Visual Explanation)



For each assignment we must check subset constraints.

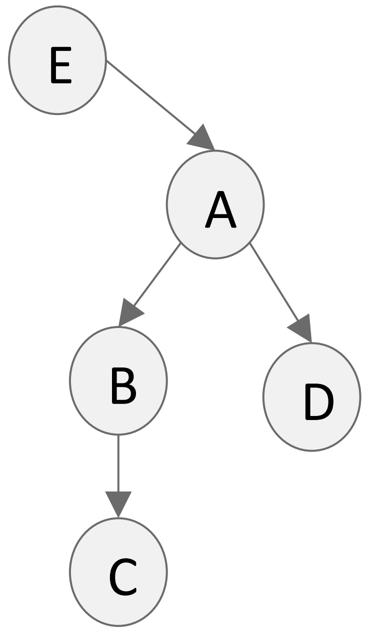
- C=B; so P(C) ⊇ P(B)
- B=A; so P(B) ⊇ P(A)
- D=A; so P(D) ⊇ P(A)

Computational Complexity (Visual Explanation)



If the points-to set of A changes, then the points-to sets of B,C, and D can change. In the worst case we must traverse the entire graph, which is $O(n^2)$ complexity.

Computational Complexity (Visual Explanation)



Since assignments can be processed in any order, we must consider the worst case where *every* assignment causes us to re-traverse the entire graph.

Consider A=E;

In the worst case there can be n re-traversals of the graph. So, $n*n^2 = O(n^3)$.

Basic Anderson's Algorithm in Atlas (Overview)

- Local Points-to Analysis Example in ~30 lines of code
  1. Assign a unique ID to each "new" allocation
  2. For each assignment, propagate the IDs on the right hand side of the assignment to the left hand side of the assignment
  3. Repeat until no IDs can be propagated (fixed point)

- Key Atlas Features:
  1. Tags (quickly locate allocation sites)
  2. Attributes (storage of points-to set information)
  3. Data Flow Edges (assignments)
  4. *Static Single Assignment* (SSA) form (captures local statement ordering)

Basic Anderson’s Algorithm in Atlas

1. Assign a unique ID to each “new” allocation
   - Let a unique ID be a unique integer for each allocation site
   - Store IDs in an Atlas Node attribute containing a set of integers
   - New allocations are tagged with XCSG.Instantiation

```
int id = 0;
LinkedList<Node> worklist = new LinkedList<Node>();

AtlasSet<Node> instantiations =
universe.nodesTaggedWithAny(XCSG.Instantiation).eval().nodes();

for(Node instantiation : instantiations){
   Set<Integer> pointsToIds = getPointsToSet(instantiation);
   pointsToIds.add(id++);
   worklist.add(instantiation);
}
```

```
// helper method to create/access points-to sets
Set<Integer> getPointsToSet(Node node){
   String P2ID = “points-to-set”;
   if(ge.hasAttr(P2ID)){
      return (HashSet<Integer>) node.getAttr(P2ID);
   } else {
      Set<Integer> pointsToIds = new HashSet<Integer>();
      node.putAttr(P2ID, pointsToIds);
      return pointsToIds;
   }
}
```

Basic Anderson's Algorithm in Atlas

2. For each assignment, propagate the IDs on the right hand side of the assignment to the left hand side of the assignment
   - Atlas represents assignments as data flow edges (from -> to)
   - LHS points-to set must be a superset of the RHS points-to set

```
void propagatePointsTo(Node from){
    Q dataFlowEdges universe.edgesTaggedWithAny(XCSG.LocalDataFlow);
    AtlasSet<Node> toNodes = dataFlowEdges.successors(Common.toQ(from)).eval().nodes();
    for(Node to : toNodes){
        Set<Integer> fromPointsToSet = getPointsToSet(from);
        Set<Integer> toPointsToSet = getPointsToSet(to);
        if(toPointsToSet.addAll(fromPointsToSet)){
            worklist.add(to);
        }
    }
}
```

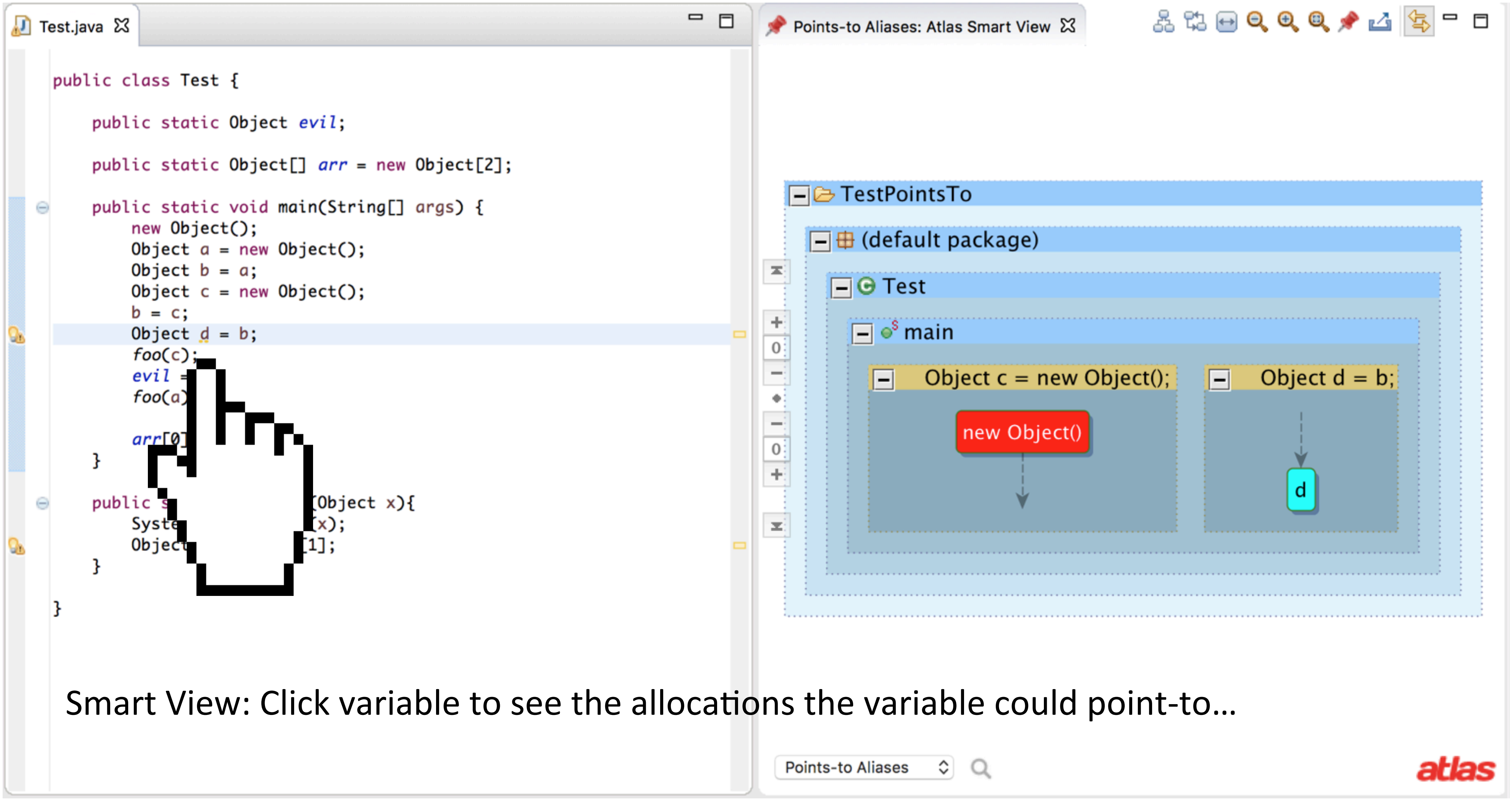
Basic Anderson’s Algorithm in Atlas

3. Repeat until no IDs can be propagated (fixed point)

```
// keep propagating allocation ids forward along assignments
// until there is nothing more to propagate
while(!worklist.isEmpty()){
   Node from = worklist.removeFirst();
   propagatePointsTo(from);
}
```

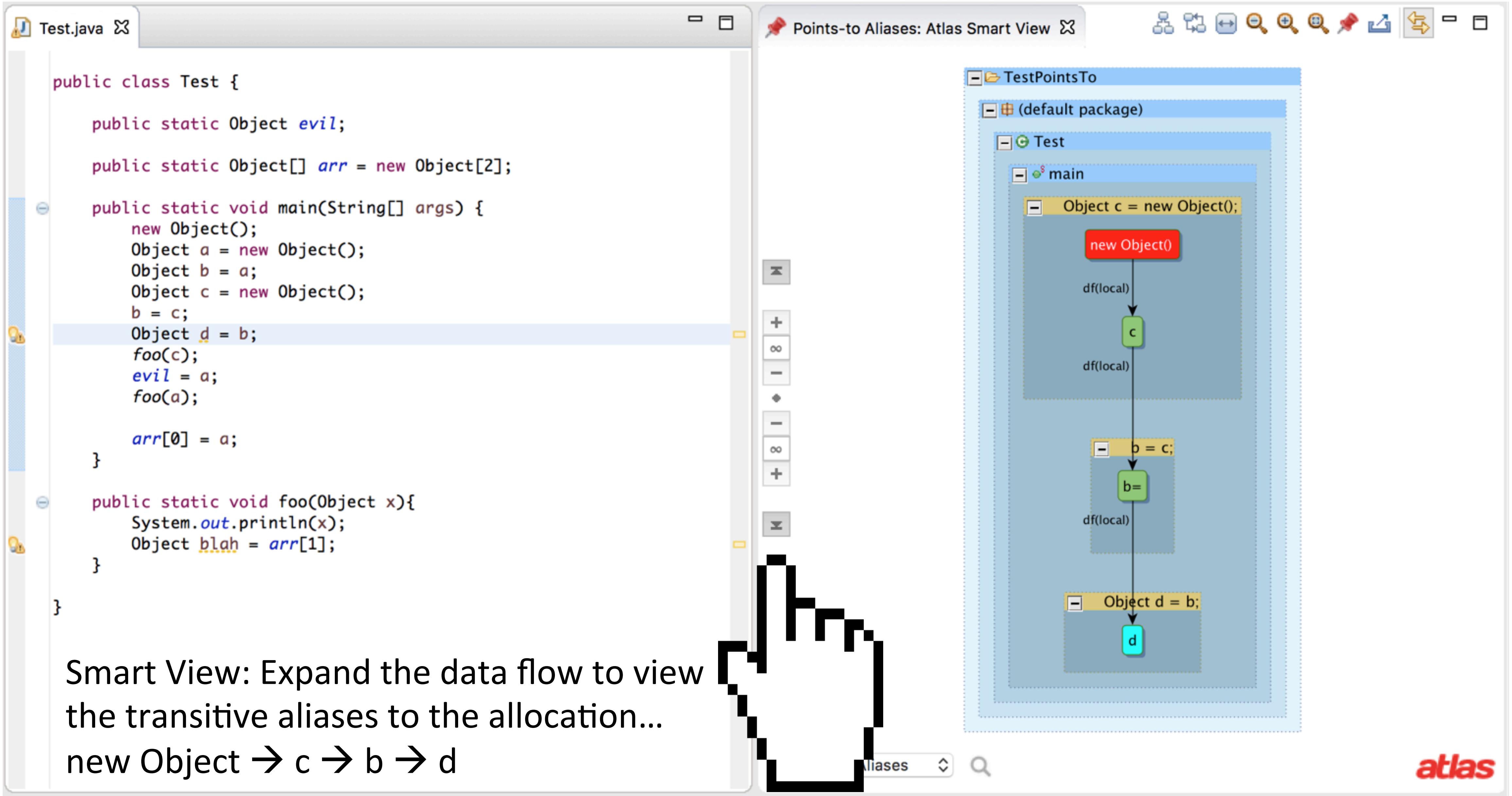
Advanced Anderson’s Algorithm in Atlas

- Consider resolving dynamic dispatches on-the-fly for interprocedural data flow
  - Variable runtime type must be known to resolve dispatch target
- Consider more efficient set storage (ex: BDDs)
- Consider array component access
- Implementation: https://github.com/EnSoftCorp/points-to-toolbox



```
public class Test {

    public static Object evil;

    public static Object[] arr = new Object[2];

    public static void main(String[] args) {
        new Object();
        Object a = new Object();
        Object b = a;
        Object c = new Object();
        b = c;
        Object d = b;
        foo(c);
        evil =
        foo(a)

        arr[0]
    }

    public s                  (Object x){
        Syste                 (x);
        Objec                 1];
    }

}
```

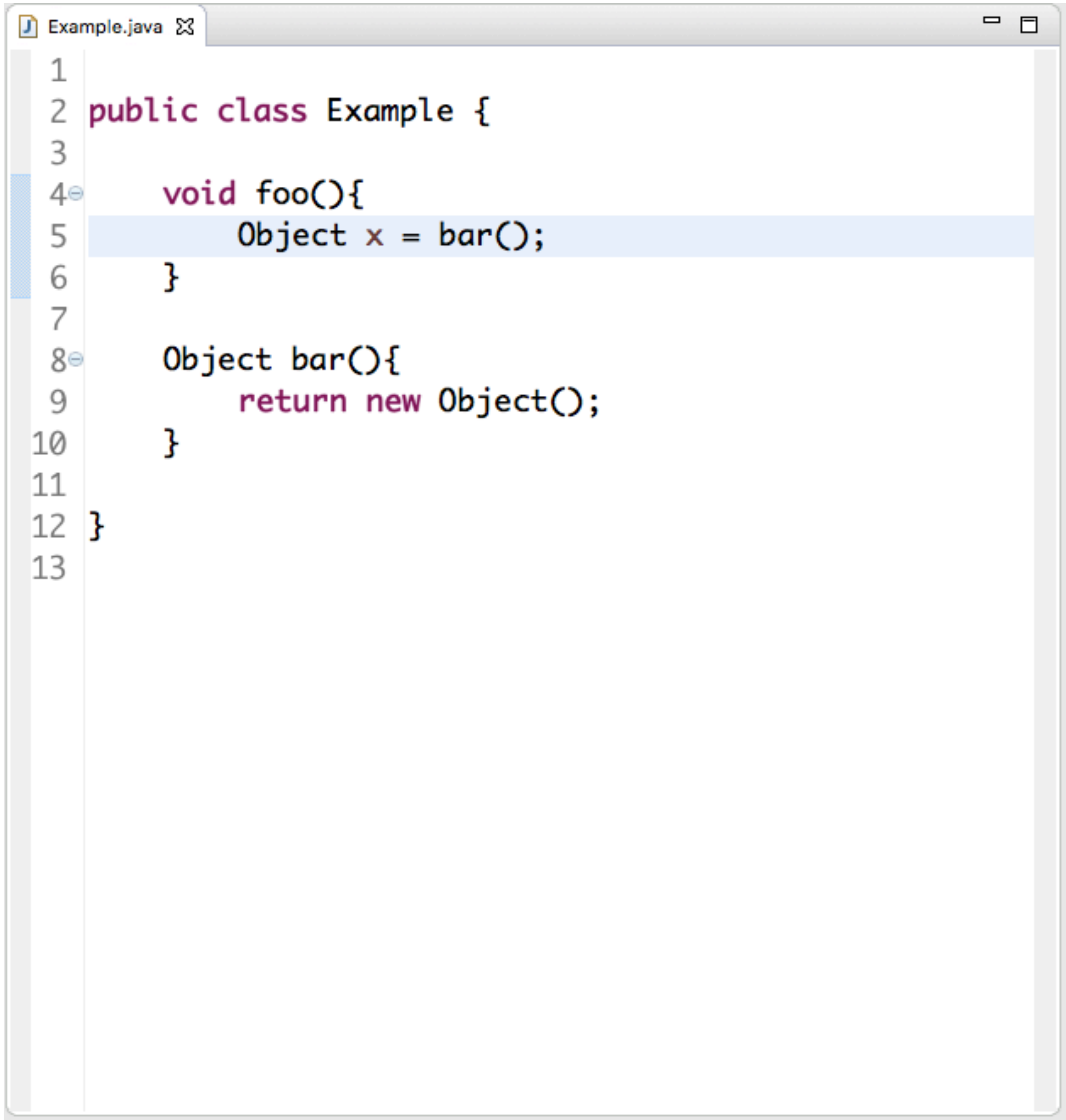
Smart View: Click variable to see the allocations the variable could point-to…


Test.java

```
public class Test {

    public static Object evil;

    public static Object[] arr = new Object[2];

    public static void main(String[] args) {
        new Object();
        Object a = new Object();
        Object b = a;
        Object c = new Object();
        b = c;
        Object d = b;
        foo(c);
        evil = a;
        foo(a);

        arr[0] = a;
    }

    public static void foo(Object x){
        System.out.println(x);
        Object blah = arr[1];
    }

}
```

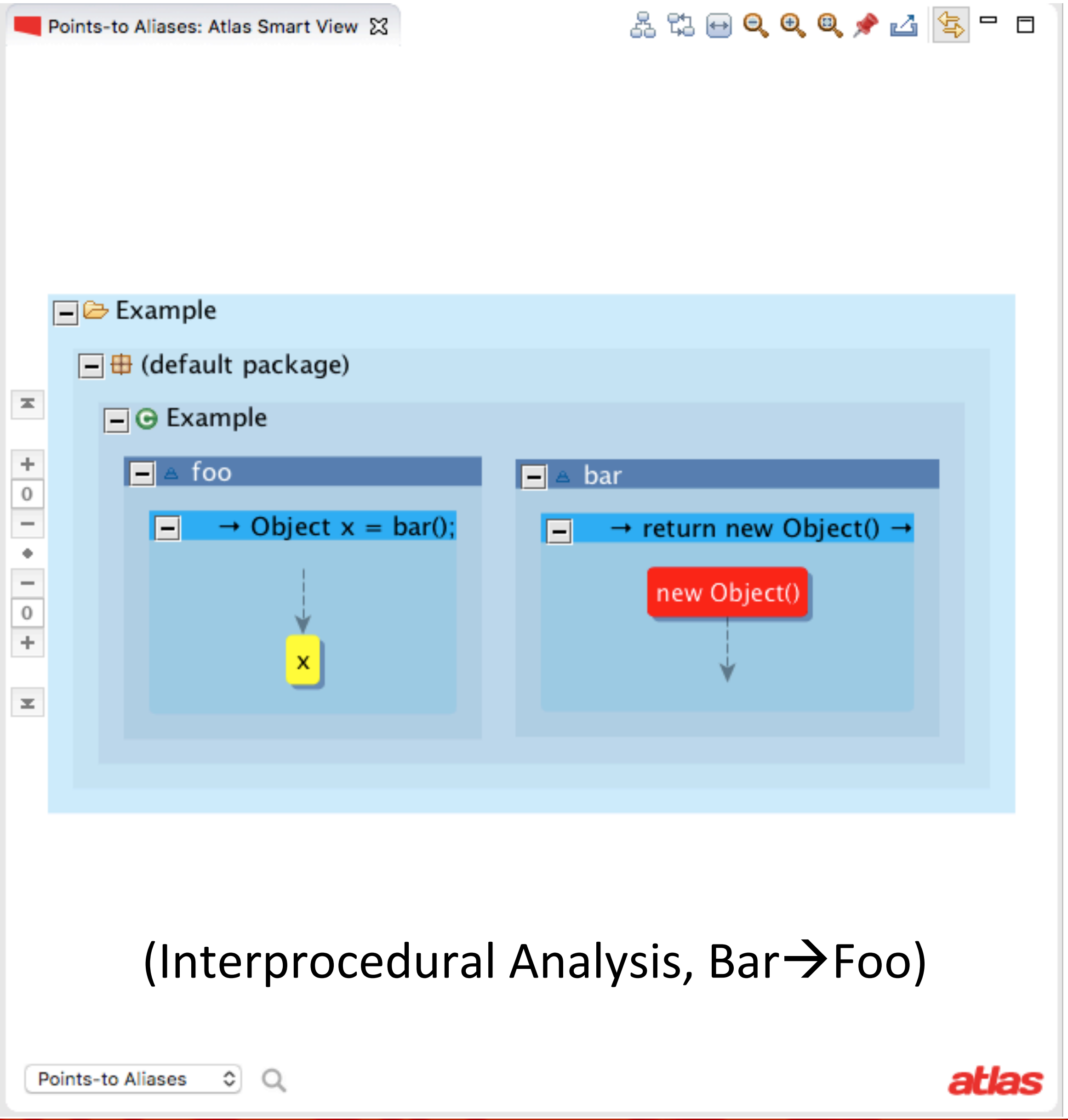
Smart View: Expand the data flow to view the transitive aliases to the allocation…

new Object → c → b → d



Example.java

```
public class Example {

    void foo(){
        Object x = bar();
    }

    Object bar(){
        return new Object();
    }

}
```



Points-to Aliases: Atlas Smart View

(Interprocedural Analysis, Bar→Foo)

A practical question

```
1.  T1 = P(O1);  - T1 is set to point to O1
2.  T2 = P(O2); - T1 is set to point to O2
3.  IF(C)
4.  T1 = T2;
5.  ELSE
6.  Deallocate(T1);
```

Points-to(T1, 6): The set of objects T1 points to at the program point 6.

What is Points-to(T1, 6)?

Ans: Points-to(T1, 6) = {O1}

Anderson's Algorithm

1. T1 = P(O1); - T1 is set to point to O1
2. T2 = P(O2); - T1 is set to point to O2
3. IF(C)
4. T1 = T2;
5. ELSE
6. Deallocate(T1);

Anderson's algorithm would give:

Points-to(T1) = {O1, O2} and,

Points-to(T2) = {O2}

How to use the Anderson's algorithm to get the points-to set at a particular point in the program?

Leveraging an idea from compiler optimization

Compiler Optimizations:

```
read(i);
j = i + 1;
k = i;
t = k +1;
```

Value numbering determines that j == t

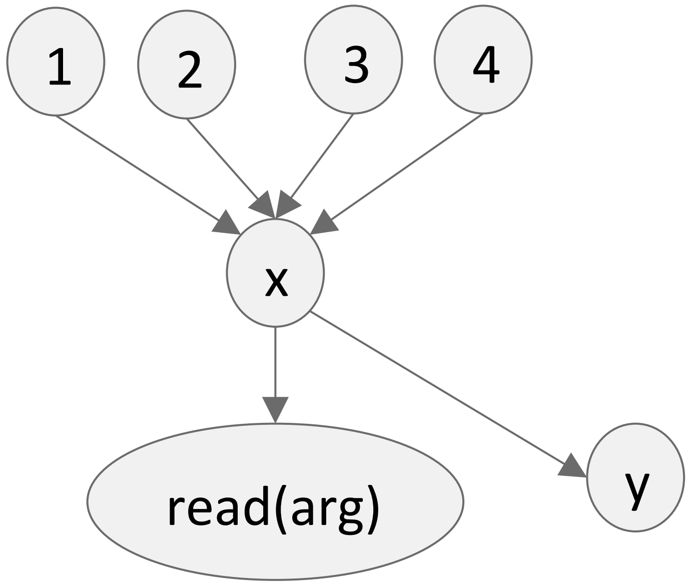
```
i = 2;
j = i * 2;
k = i + 2;
```

Constant propagation determines that j == k

Static Single Assignment (SSA) is the idea compilers use to perform such optimizations.

Without SSA

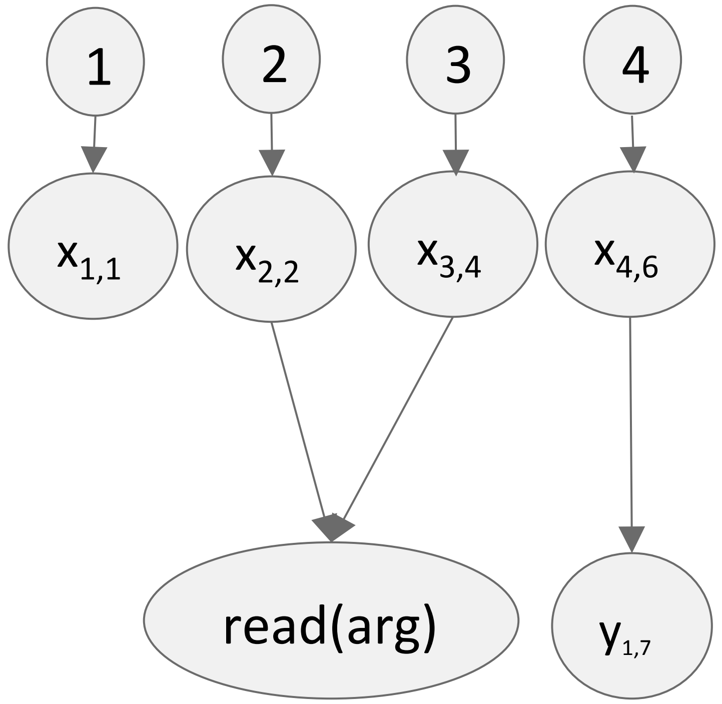
```
1.  x = 1;
2.  x = 2;
3.  if(condition)
4.      x = 3;
5.  read(x);
6.  x = 4;
7.  y = x;
```



With SSA

```
1.  x = 1;
2.  x = 2;
3.  if(condition)
4.     x = 3;
5.  read(x);
6.  x = 4;
7.  y = x;
```

1. $x_{1,1}$ = 1;
2. $x_{2,2}$ = 2;
3. if(condition)
4. $\quad x_{3,4}$ = 3;
5. read($x_{2,2},_{3,4}$);
6. $x_{4,6}$ = 4;
7. $y_{1,7}$ = $x_{4,6}$;



Transform to SSA

```
1.  T1 = P(O1);
2.  T2 = P(O2);
3.  IF(C)
4.  T1 = T2;
5.  ELSE
6.  Deallocate(T1);
```

```
1.  T11 = P(O1);
2.  T22 = P(O2);
3.  IF(C)
4.  T14 = T22;
5.  ELSE
6.  Deallocate(T11);
```

While creating new variables to enforce SSA, we can incorporate the program point information in the new variable name. That enables source correspondence. The name T14 conveys that it is the variable T1 defined at the program point 4.

A practical question – after applying SSA

1. T11 = P(O1); - T11 is set to point to O1
2. T22 = P(O2); - T21 is set to point to O2
3. IF(C)
4. T14 = T22;
5. ELSE
6. Deallocate(T11);

Points-to(T11, 6): The set of objects T11 points to at the program point 6.

Anderson algorithm will produce the answer:

Points-to(T11, 6) = {O1}