Control Flow Analysis

Static Analysis 2009

Michael I. Schwartzbach
Computer Science, University of Aarhus

Control Flow Complications

- Function pointers complicate CFG construction:
  - several functions may be invoked at a call site
  - this depends on the dataflow
  - but dataflow analysis first requires a CFG
- Same situation for other features:
  - higher-order functions
  - a class hierarchy with objects and methods
  - prototype objects with dynamic properties

Control Flow Analysis

- A control flow analysis approximates the CFG
  - conservatively computes possible functions at call sites
  - the trivial answer: *all* functions
- This is a flow-*insensitive* analysis:
  - it is based on the AST
  - the CFG is not available yet
- A subsequent analysis may use the CFG:
  - a flow-sensitive CFA may be less conservative
  - this could be iterated

CFA for The Lambda Calculus

- The pure lambda calculus

E	$\rightarrow \lambda x.E \mid$	(function definition)
	$E_1\ E_2 \mid$	(function application)
	x	(variable reference)

- Assume all λ-bound variables are distinct
- A *closure* λx abstracts the function $\lambda x.E$ in all contexts (values of free variables)
- For each call site E_1E_2 determine the possible functions for E_1 from the set $\{\lambda x_1, \lambda x_2, ..., \lambda x_n\}$

Closure Analysis

- For every AST node, v, we introduce a variable [[v]] ranging over subsets of closures
- For $\lambda id.E$ we have the constraint:

  $\{\lambda id\} \subseteq [[\lambda id.E]]$

- For E_1E_2 we have the *conditional* constraint:

  $\lambda id \in [[E_1]] \Rightarrow [[E_2]] \subseteq [[id]] \wedge [[E]] \subseteq [[E_1E_2]]$

  for every closure λid

The Cubic Framework

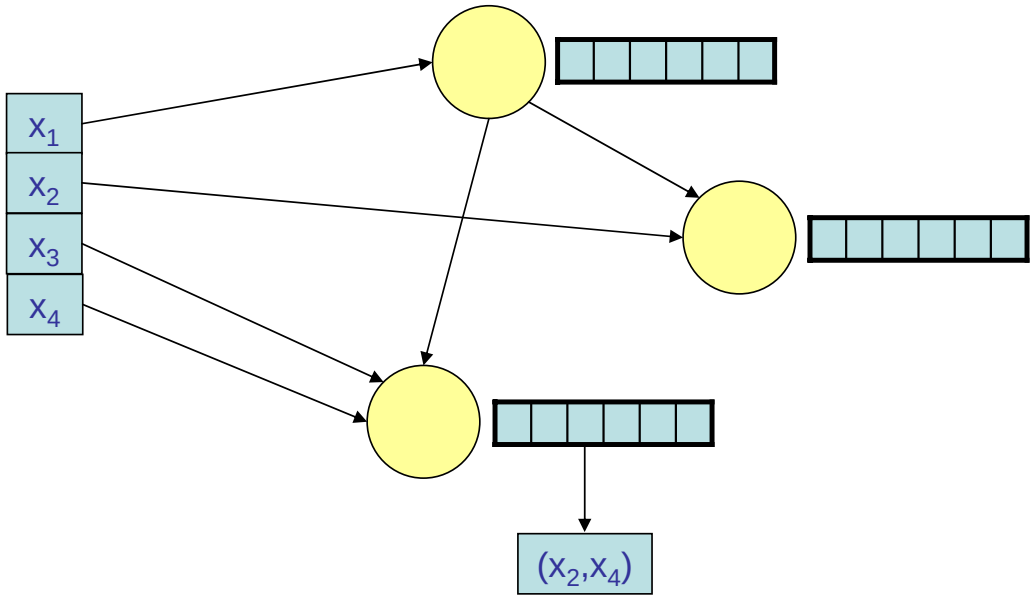
- We have a set of tokens $\{t_1, t_2, ..., t_k\}$
- We have a collection of variables $\{x_1, ..., x_n\}$ ranging over subsets of tokens
- A collection of constraints of the form:
  - $\{t\} \subseteq x$
  - $t \in x \Rightarrow y \subseteq z$
- Compute the unique minimal solution
  - this exists since solutions are closed under intersection
- An cubic time algorithm is available

The Solver Data Structure

- Each variable is mapped to a node in a DAG
- Each node has a bitvector in $\{0,1\}^k$
  - initially set to all 0's
- Each bit has a list of pairs of variables
  - used to model conditional constraints
- The DAG edges model inclusion constraints

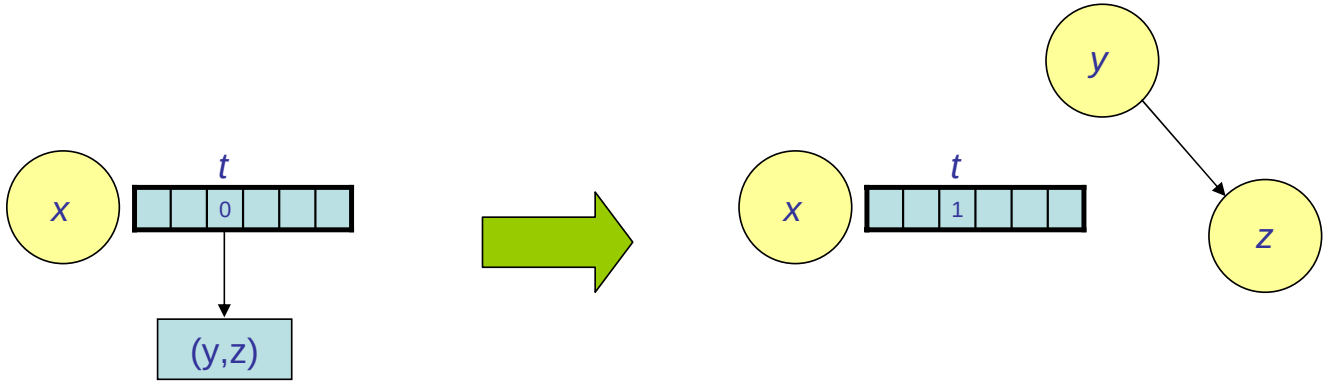
- The bitvectors will at all times directly represent the minimal solution to the constraints seen so far

An Example Graph



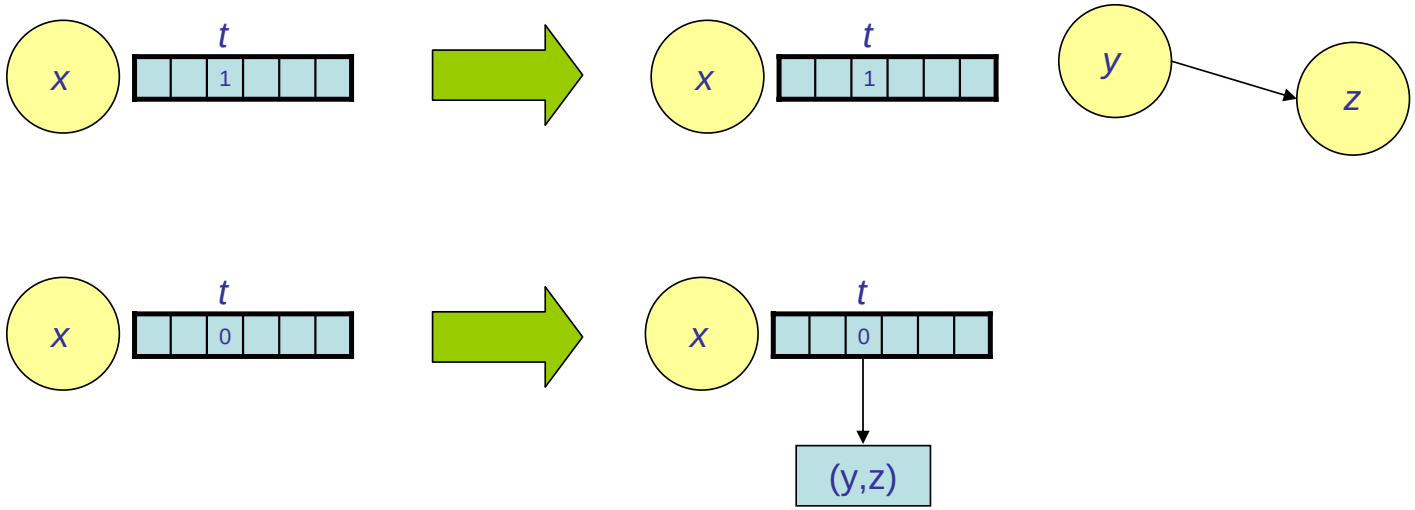
Adding Constraints (1/2)

- Constraints of the form $\{t\} \subseteq x$:
  - look up the node associated with *x*
  - set the bit corresponding to *t* to 1
  - if the list of pairs for t is not empty, then add the edges corresponding to the pairs to the DAG



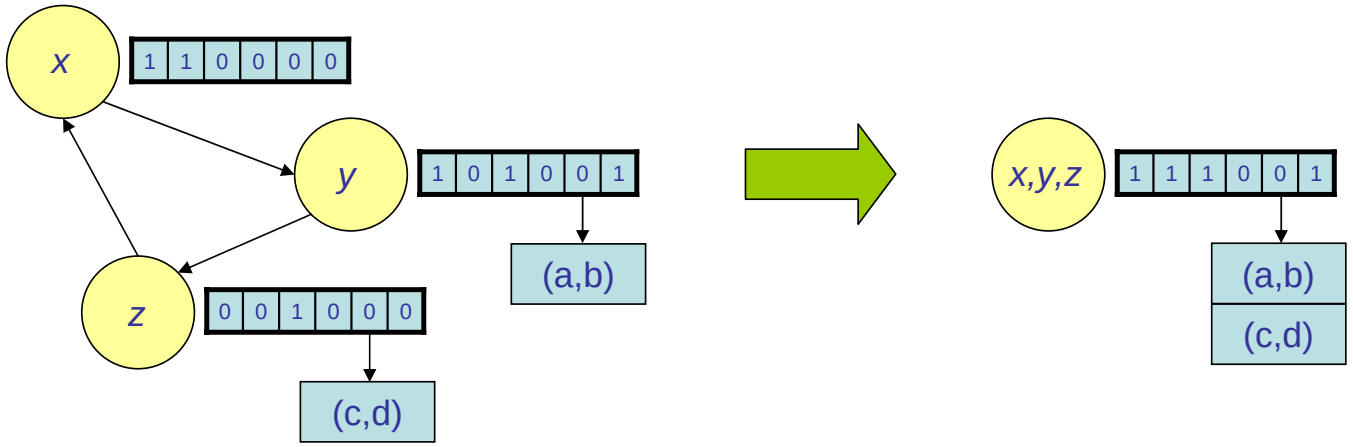
Adding Constraints (2/2)

- Constraints of the form $t \in x \Rightarrow y \subseteq z$:
  - test if the bit corresponding to *t* is 1
  - if so, add the DAG edge from *y* to *z*
  - otherwise, add (*y*,*z*) to the list of pairs for *t*



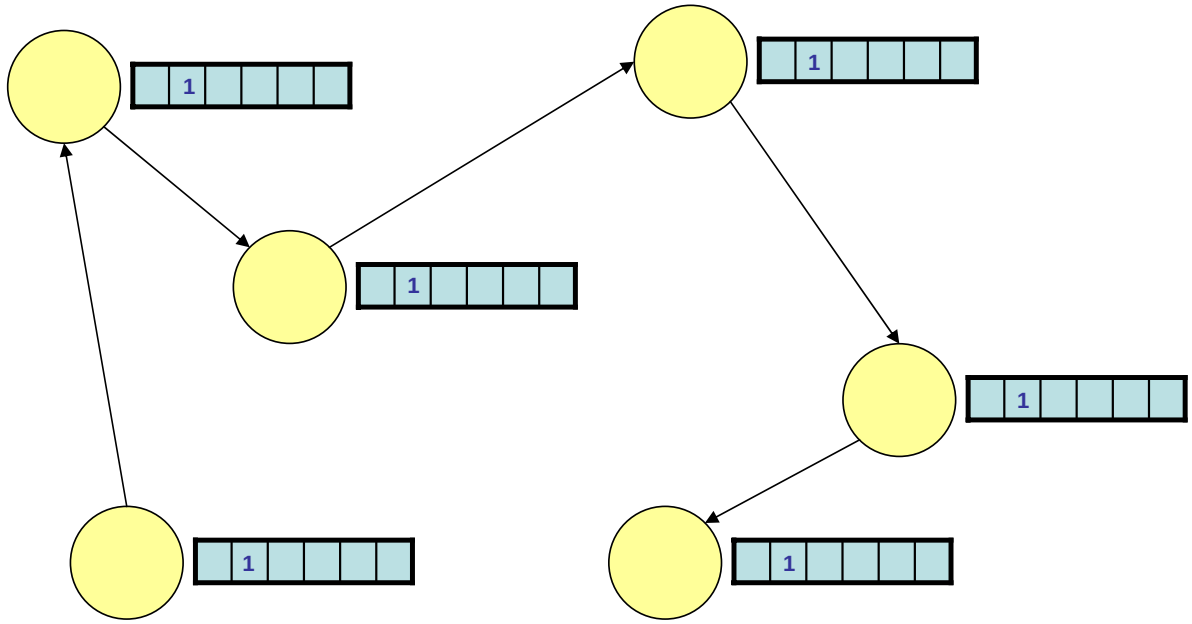
Collapse Cycles

- If a newly added edge forms a cycle:
  - merge the nodes on the cycle into a single node
  - form the union of the bitvectors
  - concatenate the lists of pairs
  - update the map from variables accordingly



Propagate Bitvectors

- Propagate the values of all newly set bits along all edges in the DAG

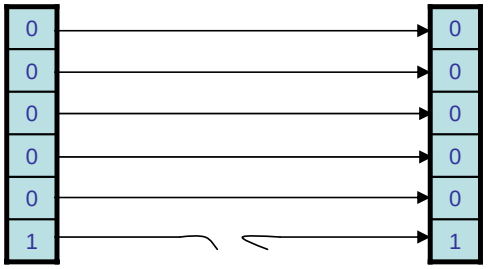


Time Complexity (1/3)

- Assume number of tokens and constraints is O(n)
  - all are proportional to the size of the program
- Merging DAG cycles:
  - at most O(n) times
  - at most O(n) nodes in each merger
  - at most O(n^2) to concatenate lists
  - at most O(n^2) to union bitvectors
  - in total at most O(n^3)
- Inserting new edges:
  - at most O(n^2) times

Time Complexity (2/3)

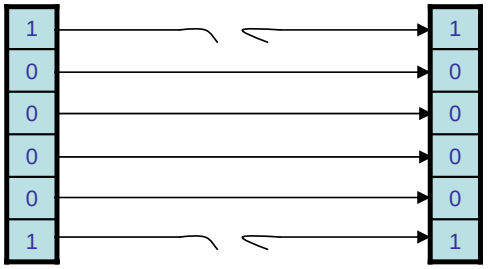
- Including constant sets:
  - at most O(n^2) times
- Propagating bits along edges:
  - imagine thin bitwires along DAG edges
  - each bit propagation burns out the wire:



  - with at most n^3 bitwires, propagation is at most O(n^3)

Time Complexity (2/3)

- Including constant sets:
  - at most O(n^2) times
- Propagating bits along edges:
  - imagine thin bitwires along DAG edges
  - each bit propagation burns out the wire:



  - with at most n^3 bitwires, propagation is at most O(n^3)

Time Complexity (3/3)

- Adding it all up, the upper bound is $O(n^3)$
- This is known as the *cubic time bottleneck*:
  - seems to be a lower bound as well
  - occurs in many different scenarios
- A special case of general set constraints:
  - defined on sets of *terms* instead of sets of tokens
  - solvable in time $O(2^{2^n})$

CFA for Function Pointers

- For a computed function call:

$$E \rightarrow (E)(E, \ldots, E)$$

  we cannot see which function is called

- A coarse but sound approximation:
  - assume any function with right number of arguments

- Use CFA to get a much better result!

CFA Constraints (1/2)

- Tokens are $\{\&id_1, \&id_2, \ldots, \&id_k\}$ for all functions
- For every AST node, v, we introduce the variable [[v]] denoting the set of functions to which v may evaluate

- For function definitions:

  $\{\&id\} \subseteq [[id]]$

- For assignments:

  $[[E]] \subseteq [[id]]$

CFA Constraints (2/2)

- For function calls:

  $$\&f \in [[E]] \Rightarrow [[E_i]] \subseteq [[a_i]] \wedge [[E']] \subseteq [[(E)(E_1, \ldots, E_n)]]$$

  for every function *f* with arguments $a_1, \ldots, a_n$ and return expression *E'*

- If we consider typable programs:
  - only generate constraints for those functions *f* for which the call would be type correct

Example Program

```
inc(i) { return i+1; }
dec(j) { return j-1; }
ide(k) { return k; }

foo(n,f) {
  var r;
  if (n==0) { f=ide; }
  r = (f)(n);
  return r;
}

main() {
  var x,y;
  x = input;
  if (x>0) { y = foo(x,inc); } else { y = foo(x,dec); }
  return y;
}
```

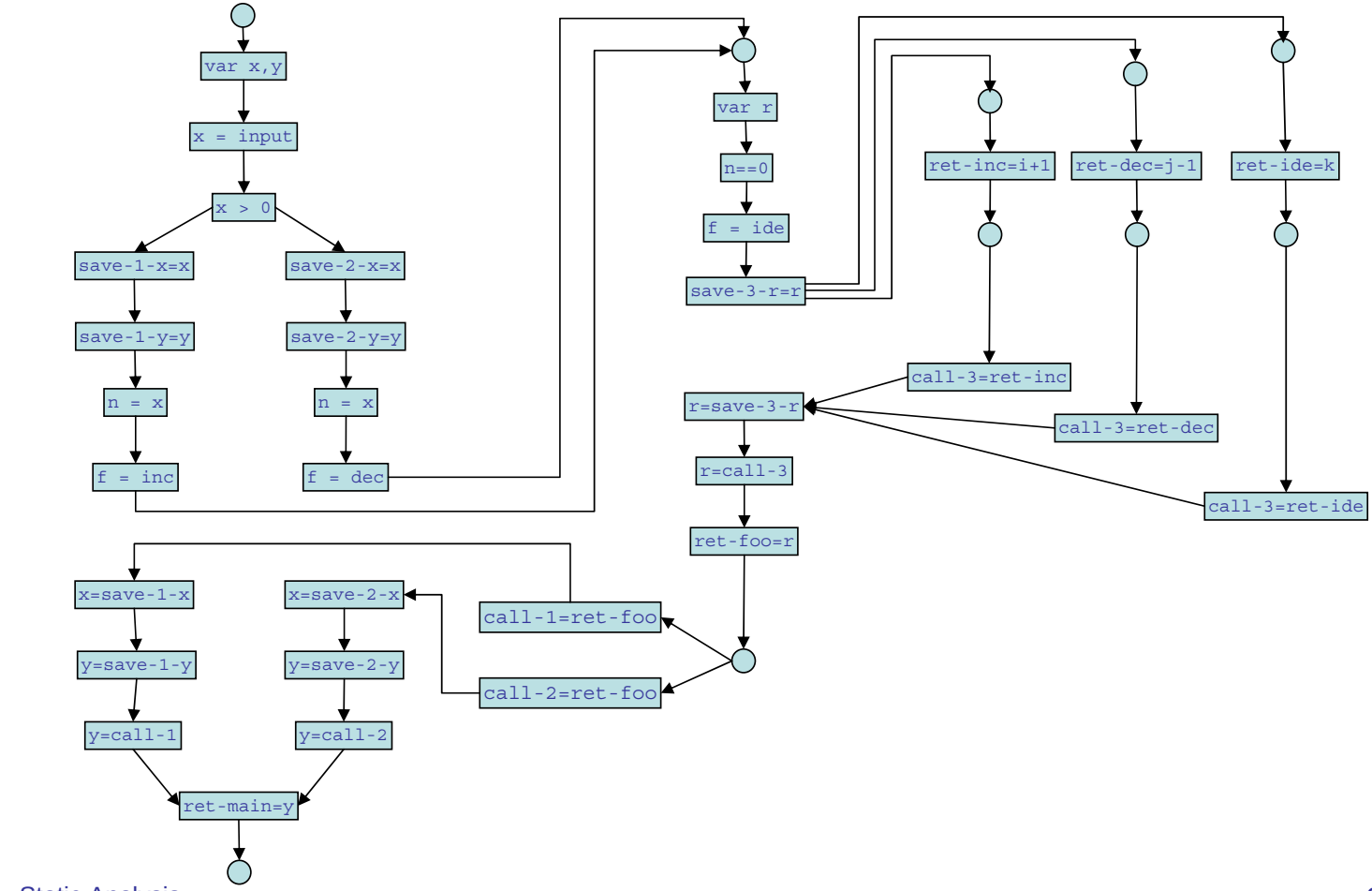
Generated Constraints

$\{\&inc\} \subseteq [[inc]]$
$\{\&dec\} \subseteq [[dec]]$
$\{\&ide\} \subseteq [[ide]]$
$[[ide]] \subseteq [[f]]$
$[[(f)(n)]] \subseteq [[r]]$
$\&inc \in [[f]] \Rightarrow [[n]] \subseteq [[i]] \wedge [[i+1]] \subseteq [[(f)(n)]]$
$\&dec \in [[f]] \Rightarrow [[n]] \subseteq [[j]] \wedge [[j-1]] \subseteq [[(f)(n)]]$
$\&ide \in [[f]] \Rightarrow [[n]] \subseteq [[k]] \wedge [[k]] \subseteq [[(f)(n)]]$
$[[input]] \subseteq [[x]]$
$[[foo(x,inc)]] \subseteq [[y]]$
$[[foo(x,dec)]] \subseteq [[y]]$
$\{\&foo\} \subseteq [[foo]]$
$\&foo \in [[foo]] \Rightarrow [[x]] \subseteq [[n]] \wedge [[inc]] \subseteq [[f]] \wedge [[(f)(n)]] \subseteq [[foo(x,inc)]]$
$\&foo \in [[foo]] \Rightarrow [[x]] \subseteq [[n]] \wedge [[dec]] \subseteq [[f]] \wedge [[(f)(n)]] \subseteq [[foo(x,dec)]]$
$\{\&main\} \subseteq [[main]]$

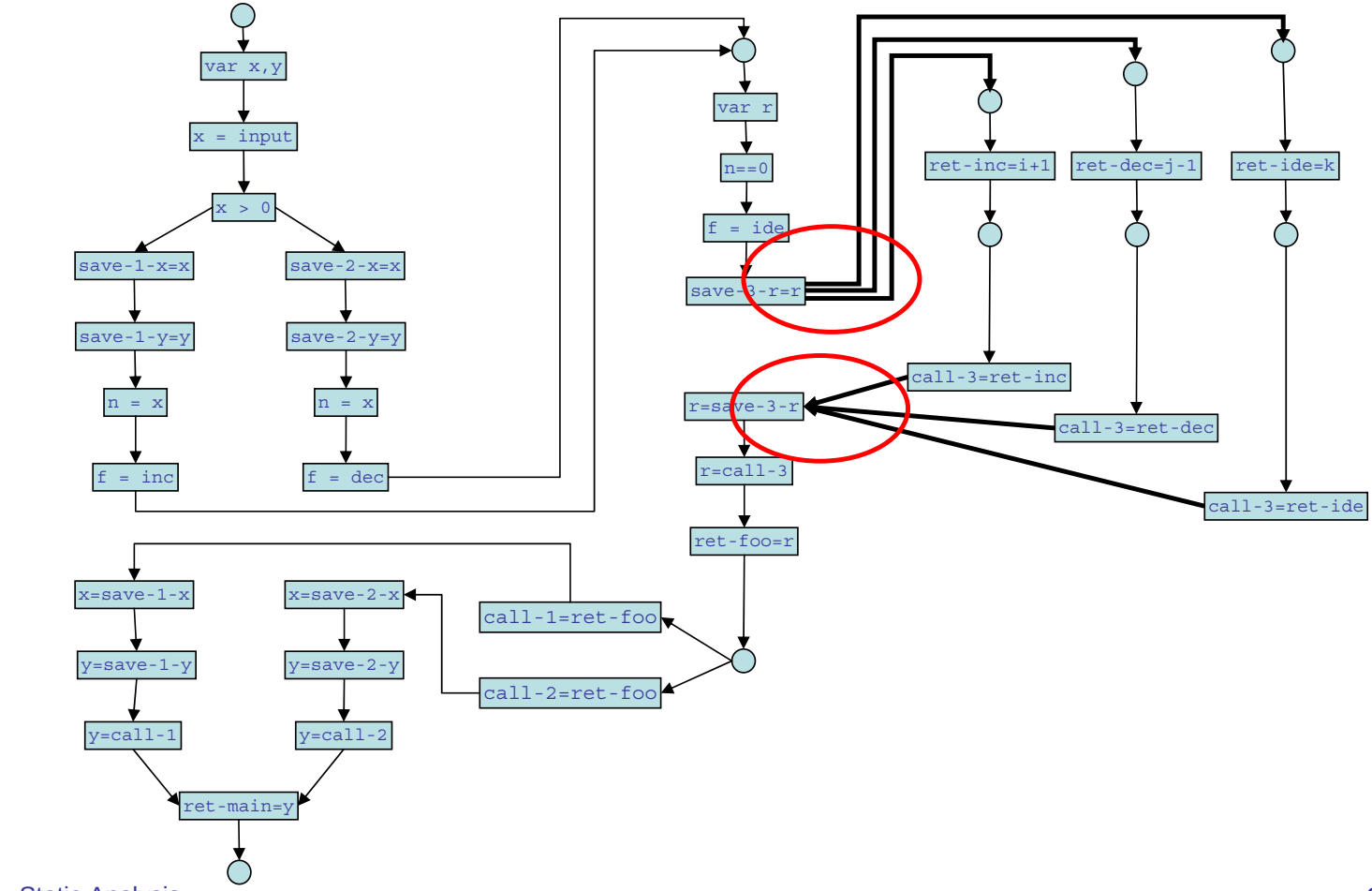
Least Solution

[[inc]] = {&ide}
[[dec]] = {&dec}
[[ide]] = {&ide}
[[f]] = {&inc, &dec, &ide}
[[foo]] = {&foo}
[[main]] = {&main}

Resulting CFG



Resulting CFG



Simple CFA for OO (1/3)

- CFA in an object-oriented language:

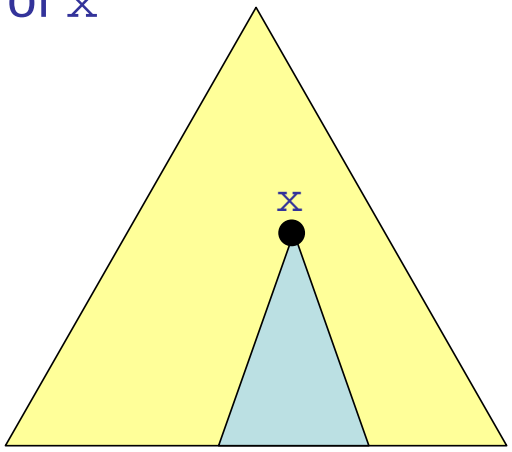
```
x.m(a,b,c)
```

- Which method implementations may be invoked?

- Full CFA is a possibility...
- But the extra structure allows simpler solutions

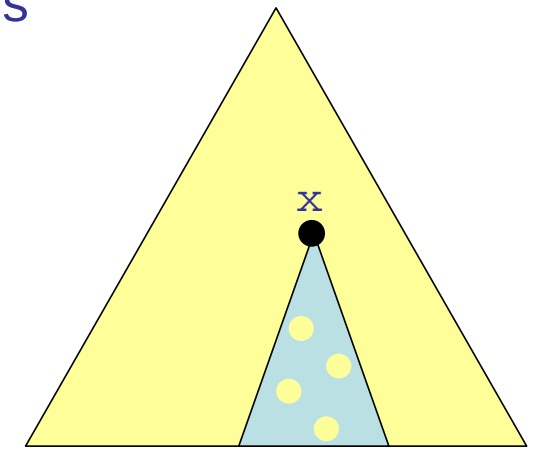
Simple CFA for OO (2/3)

- Simplest solution:
  - select all methods named `m` with three arguments
- Class Hierarchy Analysis (CHA):
  - consider only the part of the class hierarchy rooted by the declared type of `x`



Simple CFA for OO (3/3)

- Rapid Type Analysis (RTA):
  - restrict to those classes that are actually used in the program in `new` expressions



- Variable Type Analysis (VTA):
  - perform *intraprocedural* control flow analysis