Static Program Analysis

Part 8 – control flow analysis

http://cs.au.dk/~amoeller/spa/

Anders Møller & Michael I. Schwartzbach
Computer Science, Aarhus University

Agenda

- **Control flow analysis for the λ-calculus**
- The cubic framework
- Control flow analysis for TIP with function pointers
- Control flow analysis for object-oriented languages

Control flow complications

- Function pointers in TIP 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 (closures)
  - 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

控制流分析是保守地粗略估计程序的过程间控制流，也称作调用图

- Control flow analysis is usually flow-*insensitive*:
  - it is based on the AST
  - the CFG is not available yet
  - a subsequent dataflow analysis may use the CFG

- Alternative: use flow-sensitive analysis
  - potentially on-the-fly, during dataflow analysis

CFA for the lambda calculus

- The pure lambda calculus

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

- Assume all λ-bound variables are distinct
- An *abstract closure* λx abstracts the function $\lambda x.E$ in all contexts (values of free variables)
- Goal: for each call site $E_1 E_2$ determine the possible functions for E_1 from the set $\{\lambda x_1, \lambda x_2, \ldots, \lambda x_n\}$

Closure analysis

A flow-insensitive analysis that tracks function values:

- For every AST node, v, we introduce a variable ⟦v⟧ ranging over subsets of abstract closures
- For $\lambda x.E$ we have the constraint

$$\lambda x \in [\![\lambda x.E]\!]$$

- For $E_1 E_2$ we have the *conditional* constraint

$$\lambda x \in [\![E_1]\!] \Rightarrow \left([\![E_2]\!] \subseteq [\![x]\!] \wedge [\![E]\!] \subseteq [\![E_1 E_2]\!]\right)$$

  for every function $\lambda x.E$

Agenda

- Control flow analysis for the λ-calculus
- **The cubic framework**
- Control flow analysis for TIP with function pointers
- Control flow analysis for object-oriented languages

The cubic framework

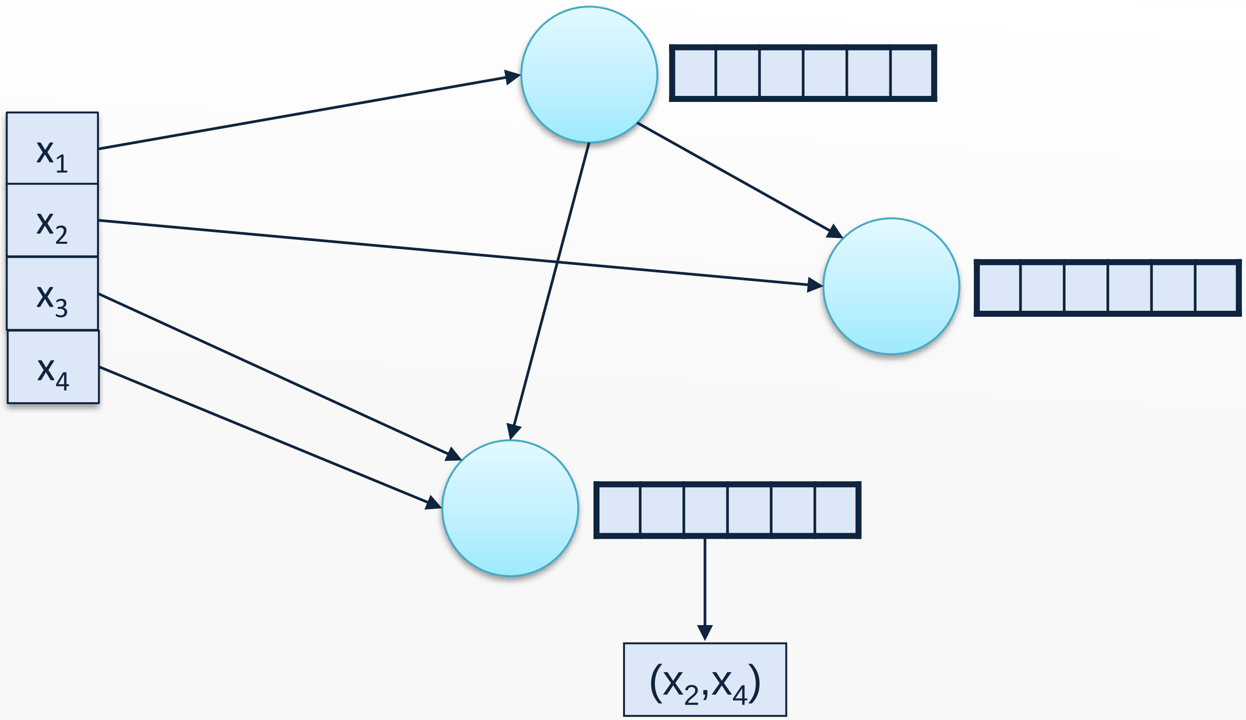
- We have a set of tokens $\{t_1, t_2, \ldots, t_k\}$
- We have a collection of variables $\{x_1, \ldots, x_n\}$ ranging over subsets of tokens
- A collection of constraints of these forms:
  - $t \in x$
  - $t \in x \Rightarrow y \subseteq z$
- Compute the unique minimal solution
  - this exists since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

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

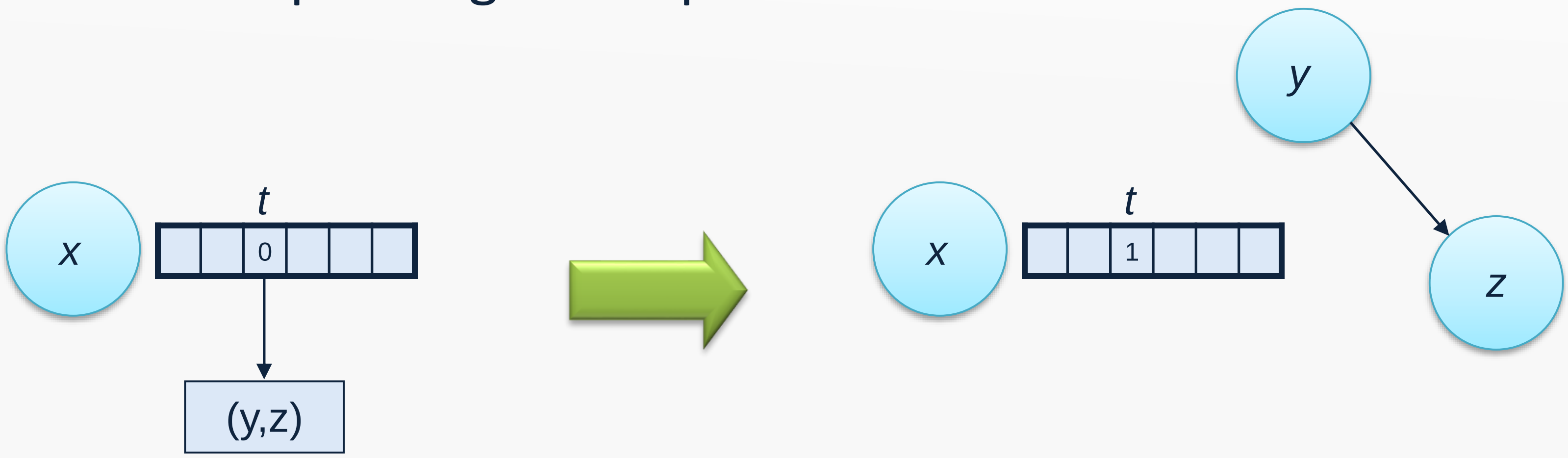
- The bitvectors will at all times directly represent the minimal solution to the constraints seen so far

An example graph



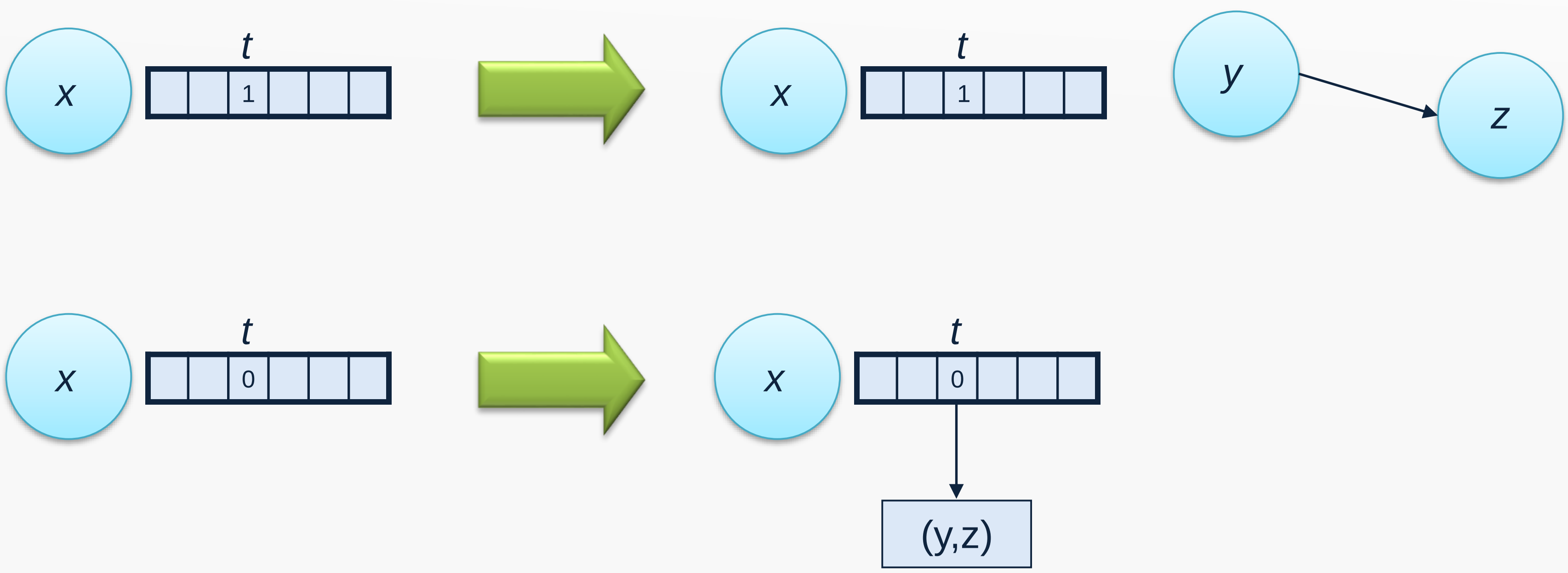
Adding constraints (1/2)

- Constraints of the form $t \in x$:
  - 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



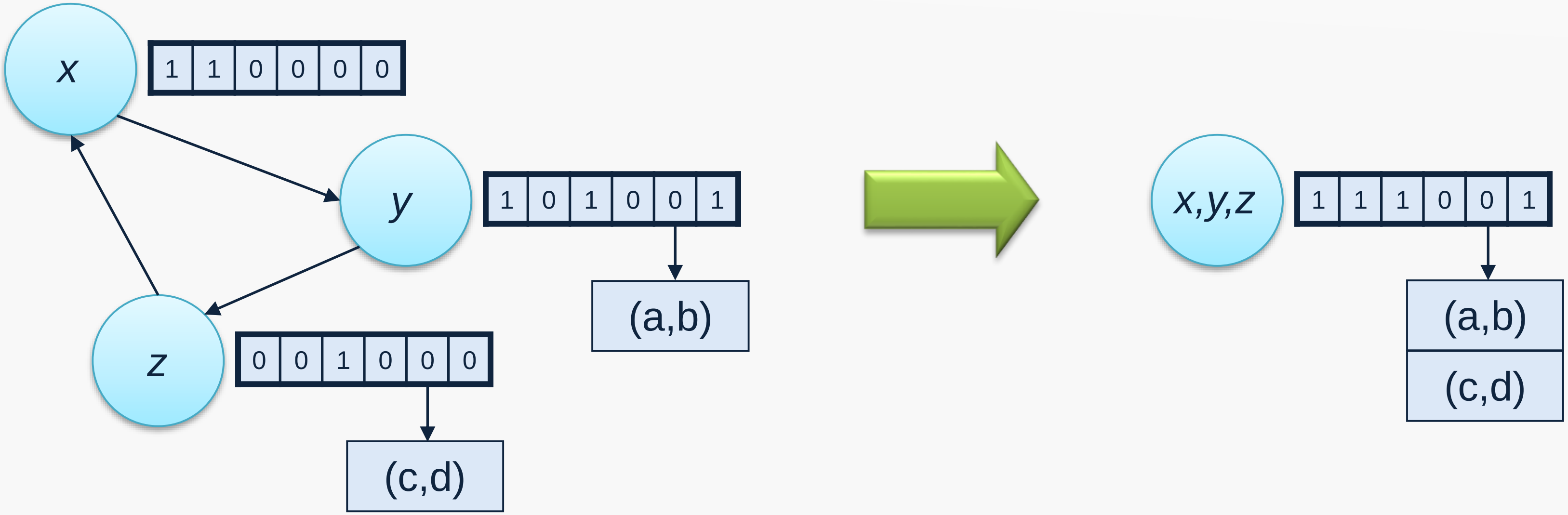
Adding constraints (2/2)

- 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*



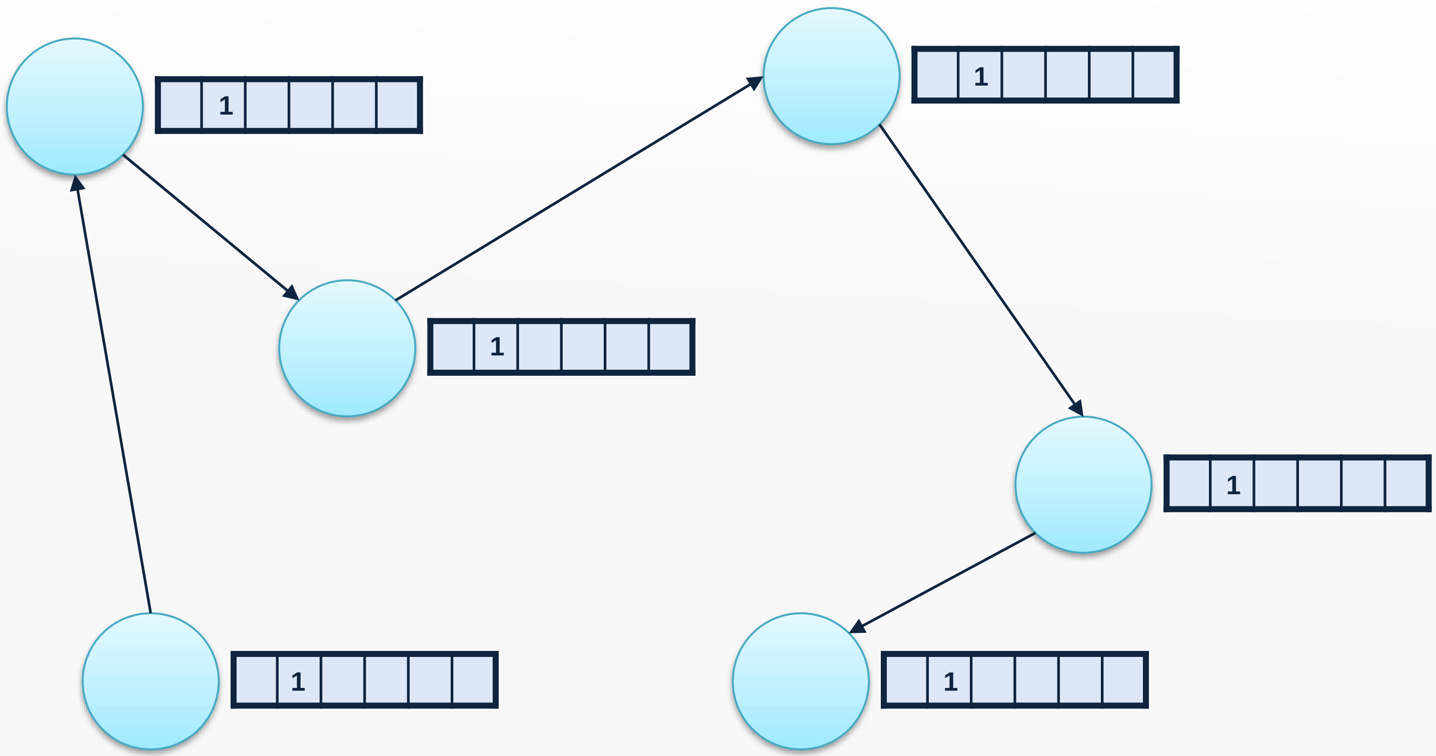
Collapse cycles

- 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



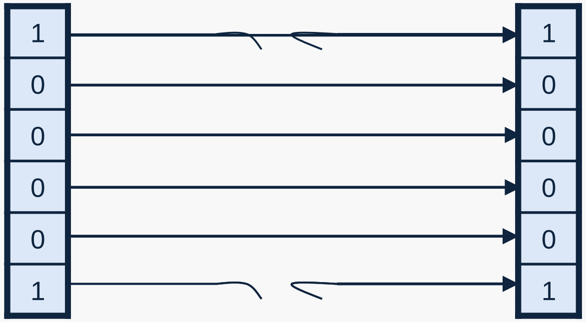
Propagate bitvectors

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



Time complexity (1/2)

- O(n) functions and O(n) applications, with program size n
- O(n) singleton constraints, O(n^2) conditional constraints
- O(n) nodes, O(n^2) edges, O(n) bits per node
- Total time for bitvector propagation: O(n^3)
- Total time for collapsing cycles: O(n^3)
- Total time for handling lists of pairs: O(n^3)



Time complexity (2/2)

- Adding it all up, the upper bound is $O(n^3)$
- This is known as the *cubic time bottleneck*:
  - occurs in many different scenarios
  - but $O(n^3/\log n)$ is possible…
- A special case of general set constraints:
  - defined on sets of *terms* instead of sets of tokens
  - solvable in time $O(2^{2^n})$

Agenda

- Control flow analysis for the λ-calculus
- The cubic framework
- **Control flow analysis for TIP with function pointers**
- Control flow analysis for object-oriented languages

CFA for TIP with function pointers

- For a computed function call

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

  we cannot immediately 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 all functions $\{f_1, f_2, \ldots, f_k\}$
- For every AST node, v, we introduce the variable $[\![v]\!]$ denoting the set of functions to which v may evaluate

- For function definitions $f(\ldots)\{\ldots\}$:

$$f \in [\![f]\!]$$

- For assignments $x = E$:

$$[\![E]\!] \subseteq [\![x]\!]$$

CFA constraints (2/2)

- For **direct** function calls $f(E_1, \ldots, E_n)$:

  $[\![E_i]\!] \subseteq [\![a_i]\!]$ for i=1,…,n $\wedge$ $[\![E']\!] \subseteq [\![f(E_1, \ldots, E_n)]\!]$

  where f has arguments $a_1, \ldots, a_n$

  and return expression E'

- For **computed** function calls $(E)(E_1, \ldots, E_n)$:

  $f \in [\![E]\!] \Rightarrow ([\![E_i]\!] \subseteq [\![a_i]\!]$ for i=1,…,n $\wedge$ $[\![E']\!] \subseteq [\![(E)(E_1, \ldots, E_n)]\!])$

  for every function f with arguments $a_1, \ldots, a_n$

  and return expression E'

  – If we consider typable programs only:

    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;
}
```

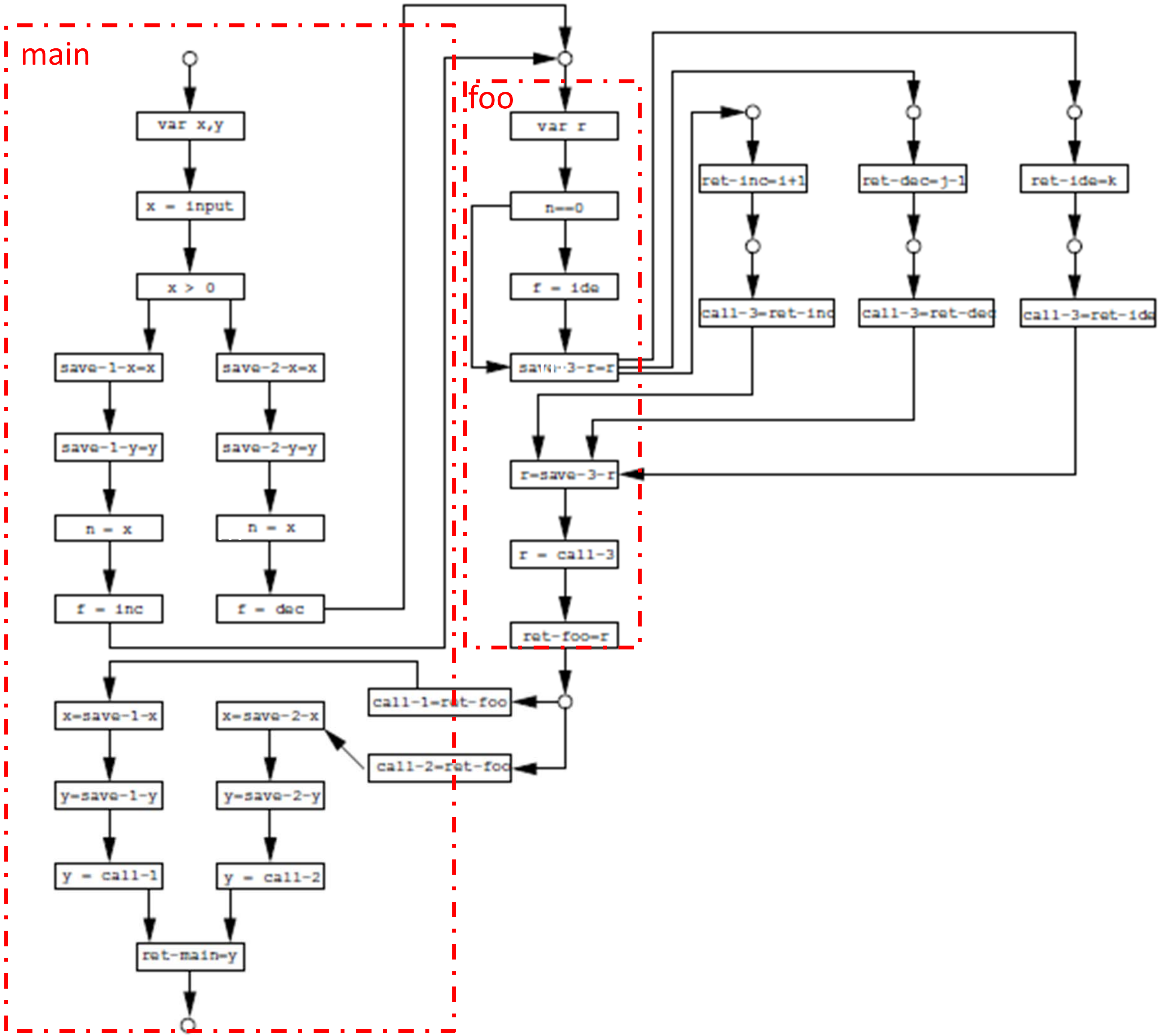
Generated constraints

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

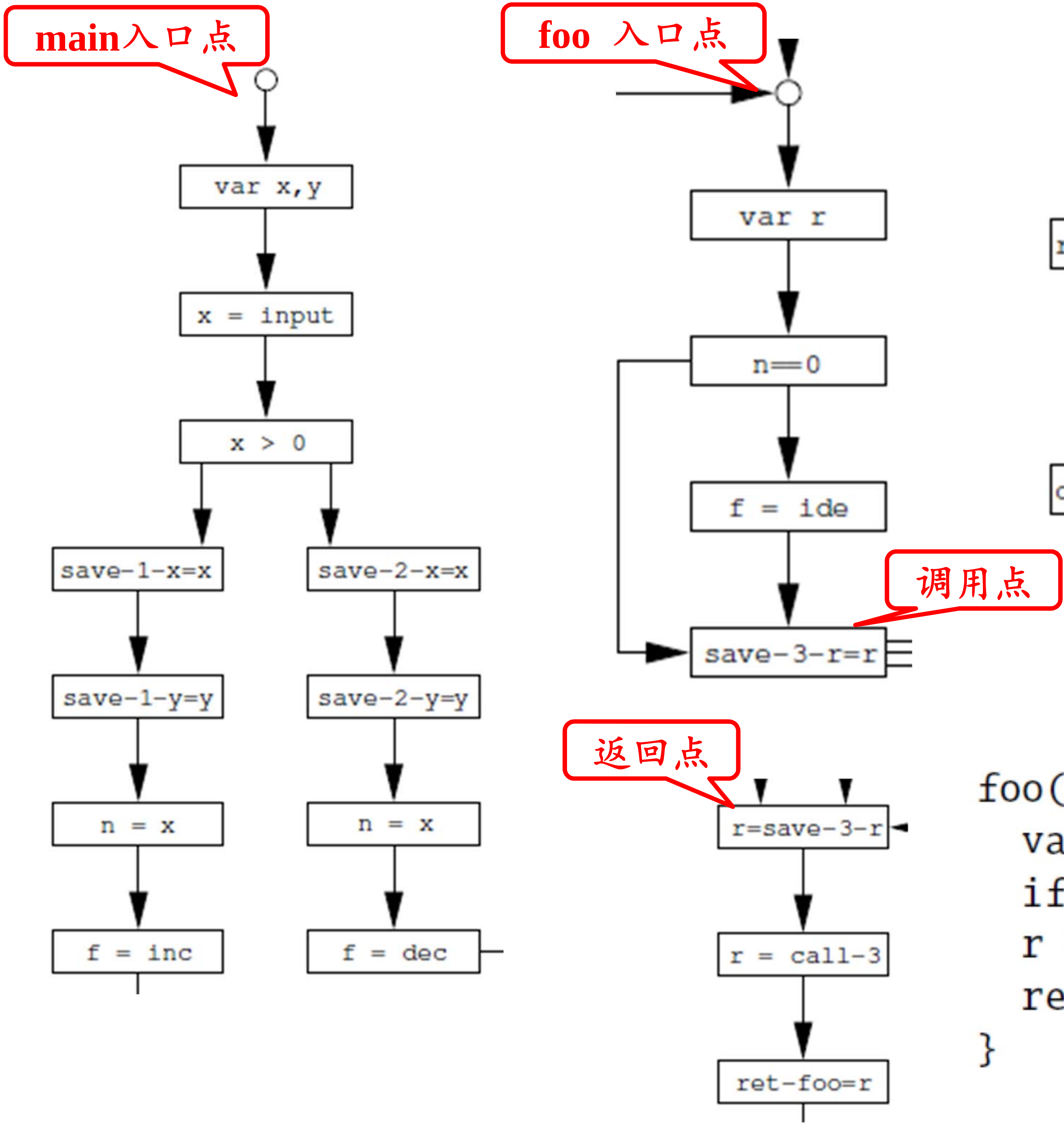
Least solution

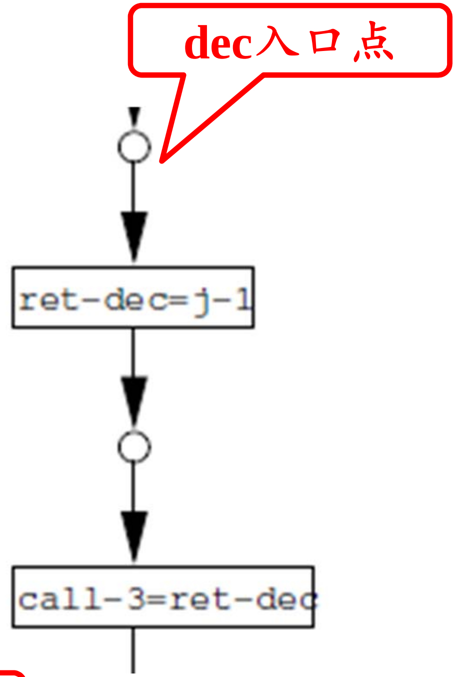
$$[\![\mathtt{inc}]\!] = \{\mathtt{inc}\}$$
$$[\![\mathtt{dec}]\!] = \{\mathtt{dec}\}$$
$$[\![\mathtt{ide}]\!] = \{\mathtt{ide}\}$$
$$[\![\mathtt{f}]\!] = \{\mathtt{inc}, \mathtt{dec}, \mathtt{ide}\}$$
$$[\![\mathtt{foo}]\!] = \{\mathtt{foo}\}$$
$$[\![\mathtt{main}]\!] = \{\mathtt{main}\}$$

With this information, we can construct the call edges and return edges in the interprocedural CFG



main
foo
var x,y
x = input
x > 0
save-1-x=x
save-1-y=y
n = x
f = inc
save-2-x=x
save-2-y=y
n = x
f = dec
var r
n==0
f = ide
save-3-r=r
r=save-3-r
r = call-3
ret-foo=r
ret-inc=i+1
call-3=ret-inc
ret-dec=j-1
call-3=ret-dec
ret-ide=k
call-3=ret-ide
call-1=ret-foo
call-2=ret-foo
x=save-1-x
y=save-1-y
y = call-1
x=save-2-x
y=save-2-y
y = call-2
ret-main=y






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

Agenda

- Control flow analysis for the λ-calculus
- The cubic framework
- Control flow analysis for TIP with function pointers
- **Control flow analysis for object-oriented languages**

Simple CFA for OO (1/3)

- CFA in an object-oriented language:

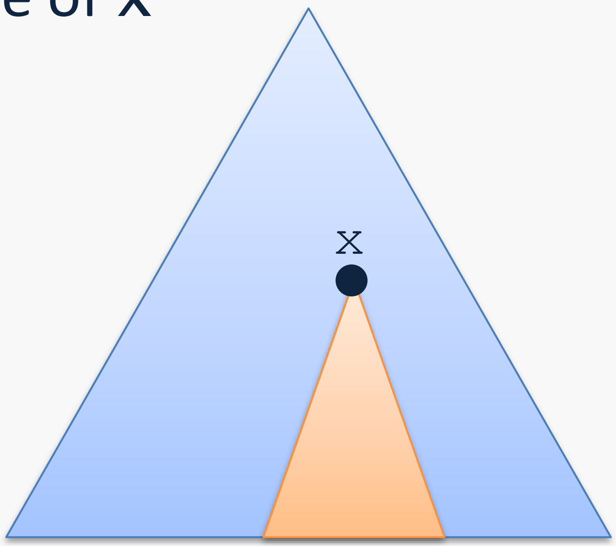
```
x.m(a,b,c)
```

- Which method implementations may be invoked?

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

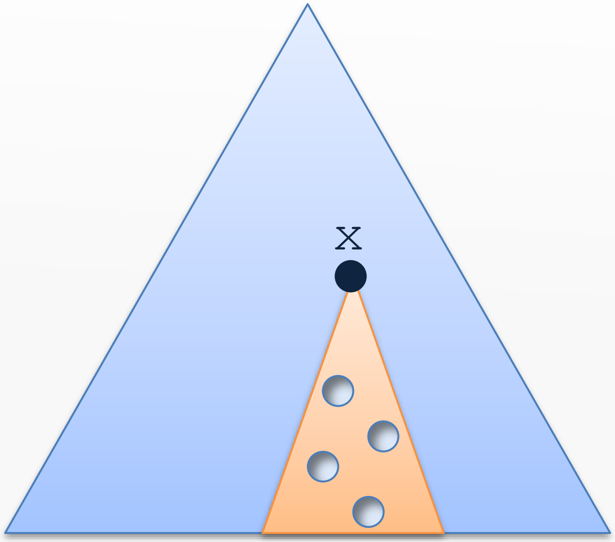
Simple CFA for OO (2/3)

- 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