Control Flow Analysis

Yu Zhang

Most content comes from http://cs.au.dk/~amoeller/spa/

http://staff.ustc.edu.cn/~yuzhang/pldpa

Agenda

- **Control flow analysis for TIP with first-class functions**
- Control flow analysis for the λ-calculus
- The cubic framework
- Control flow analysis for object-oriented languages

TIP with first-class functions

```
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;
}
```

Control Flow Complications

- First-class functions 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, e.g.
  - Function values with free variables (closures)
  - A class hierarchy with objects and methods
  - Prototype objects with dynamic properties

Control Flow Analysis

- A control flow analysis approximates the call graph
  - Conservatively computes possible functions at call sites
  - The trivial answer: *all* functions

- Control flow analysis is usually flow-*insensitive*:
  - It is based on the AST
  - The call graph can be used for an interprocedural CFG
  - A subsequent dataflow analysis may use the CFG

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

CFA for TIP with first-class functions

- For a computed function call

  $E(E_1, \ldots, E_n)$

  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

- 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

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

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

  where f is a function with arguments $a_1, \ldots, a_n$ and return expression E'
- For **computed** function calls $E(E_1, \ldots, E_n)$:

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

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

  - If consider **typable** programs only:

    Only generate constraints for those functions f for which the call would be type correct

Generated Constraints

```
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;
}
```

$\mathtt{inc} \in [\![\mathtt{inc}]\!]$

$\mathtt{dec} \in [\![\mathtt{dec}]\!]$

$\mathtt{ide} \in [\![\mathtt{ide}]\!]$

$[\![\mathtt{ide}]\!] \subseteq [\![\mathtt{f}]\!]$

$[\![\mathtt{f(n)}]\!] \subseteq [\![\mathtt{r}]\!]$

$\mathtt{inc} \in [\![\mathtt{f}]\!] \Rightarrow [\![\mathtt{n}]\!] \subseteq [\![\mathtt{i}]\!] \wedge [\![\mathtt{i+1}]\!] \subseteq [\![\mathtt{f(n)}]\!]$

$\mathtt{dec} \in [\![\mathtt{f}]\!] \Rightarrow [\![\mathtt{n}]\!] \subseteq [\![\mathtt{j}]\!] \wedge [\![\mathtt{j-1}]\!] \subseteq [\![\mathtt{f(n)}]\!]$

$\mathtt{ide} \in [\![\mathtt{f}]\!] \Rightarrow [\![\mathtt{n}]\!] \subseteq [\![\mathtt{k}]\!] \wedge [\![\mathtt{k}]\!] \subseteq [\![\mathtt{f(n)}]\!]$

$[\![\mathtt{input}]\!] \subseteq [\![\mathtt{x}]\!]$

$[\![\mathtt{foo(x,inc)}]\!] \subseteq [\![\mathtt{y}]\!]$

$[\![\mathtt{foo(x,dec)}]\!] \subseteq [\![\mathtt{y}]\!]$

$\mathtt{foo} \in [\![\mathtt{foo}]\!]$

$\mathtt{foo} \in [\![\mathtt{foo}]\!] \Rightarrow [\![\mathtt{x}]\!] \subseteq [\![\mathtt{n}]\!] \wedge [\![\mathtt{inc}]\!] \subseteq [\![\mathtt{f}]\!] \wedge [\![\mathtt{r}]\!] \subseteq [\![\mathtt{foo(x,inc)}]\!]$

$\mathtt{foo} \in [\![\mathtt{foo}]\!] \Rightarrow [\![\mathtt{x}]\!] \subseteq [\![\mathtt{n}]\!] \wedge [\![\mathtt{dec}]\!] \subseteq [\![\mathtt{f}]\!] \wedge [\![\mathtt{r}]\!] \subseteq [\![\mathtt{foo(x,dec)}]\!]$

assuming we do not use the special rule for direct calls

$\mathtt{main} \in [\![\mathtt{main}]\!]$

(At each call we only consider functions with matching number of parameters)

Least Solution

$[\![\mathtt{inc}]\!] = \{\mathtt{inc}\}$

$[\![\mathtt{dec}]\!] = \{\mathtt{dec}\}$

$[\![\mathtt{ide}]\!] = \{\mathtt{ide}\}$

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

$[\![\mathtt{foo}]\!] = \{\mathtt{foo}\}$

$[\![\mathtt{main}]\!] = \{\mathtt{main}\}$

(the solution is the empty set for the remaining constraint variables)

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

Agenda

- Control flow analysis for TIP with first-class functions
- **Control flow analysis for the λ-calculus**
- The cubic framework
- Control flow analysis for object-oriented languages

CFA for the Lambda Calculus

- The pure lambda calculus

$$
\begin{array}{lll}
E & \to \lambda x.E & \text{(function definition)} \\
 & \mid\ E_1\, E_2 & \text{(function application)} \\
 & \mid\ x & \text{(variable reference)}
\end{array}
$$

- 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 $\{\lambda x_1, \lambda x_2, ..., \lambda x_n\}$

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

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

for every function $\lambda x.E$

Agenda

- Control flow analysis for TIP with first-class functions
- Control flow analysis for the λ-calculus
- **The cubic framework**
- Control flow analysis for object-oriented languages

The Cubic Framework

- We have a set of tokens $\{t_1, t_2, ..., t_k\}$
- We have a collection of variables $\{x_1, ..., x_n\}$ whose values range 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!

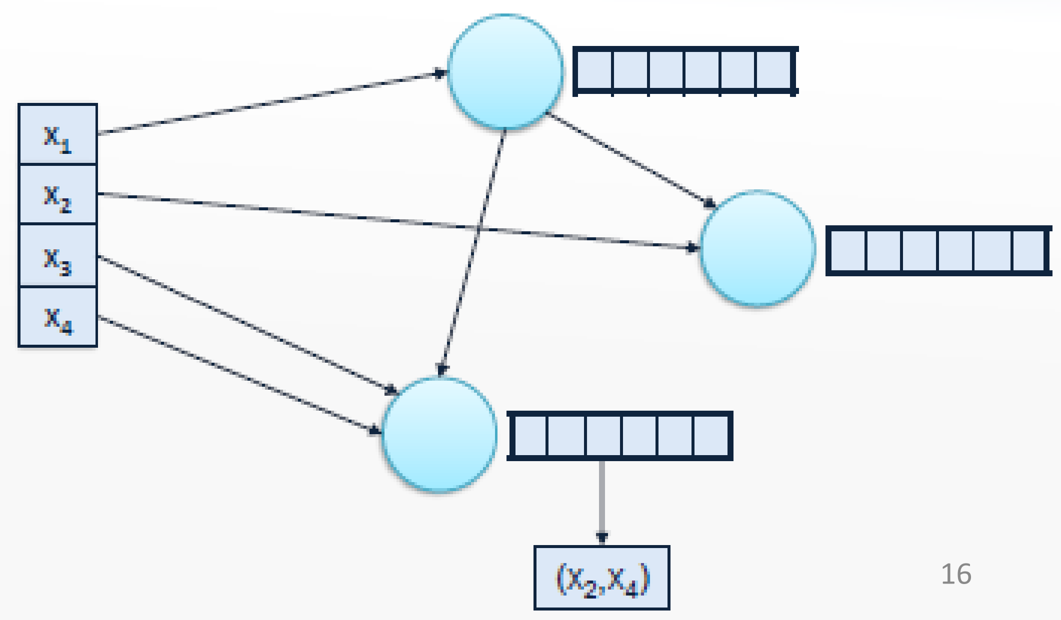
The Solver Data Structure

- 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

▪ $t \in x$

▪ $t \in x \Rightarrow y \subseteq z$

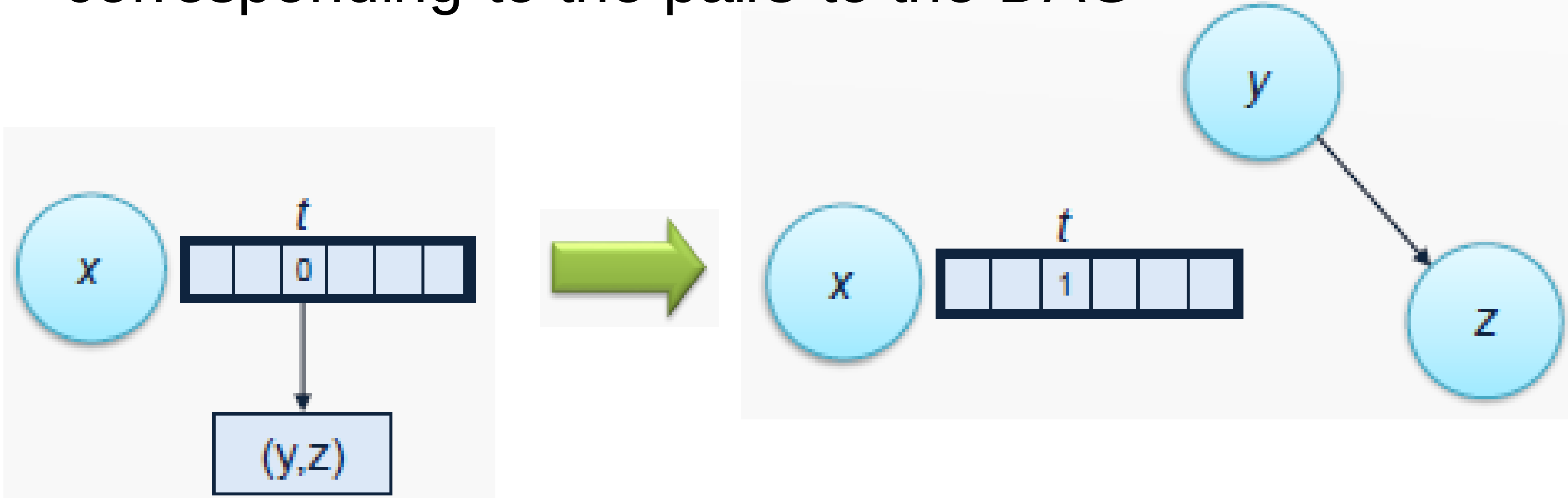
- The bitvectors will at all times directly represent the minimal solution to the constraints seen so far



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

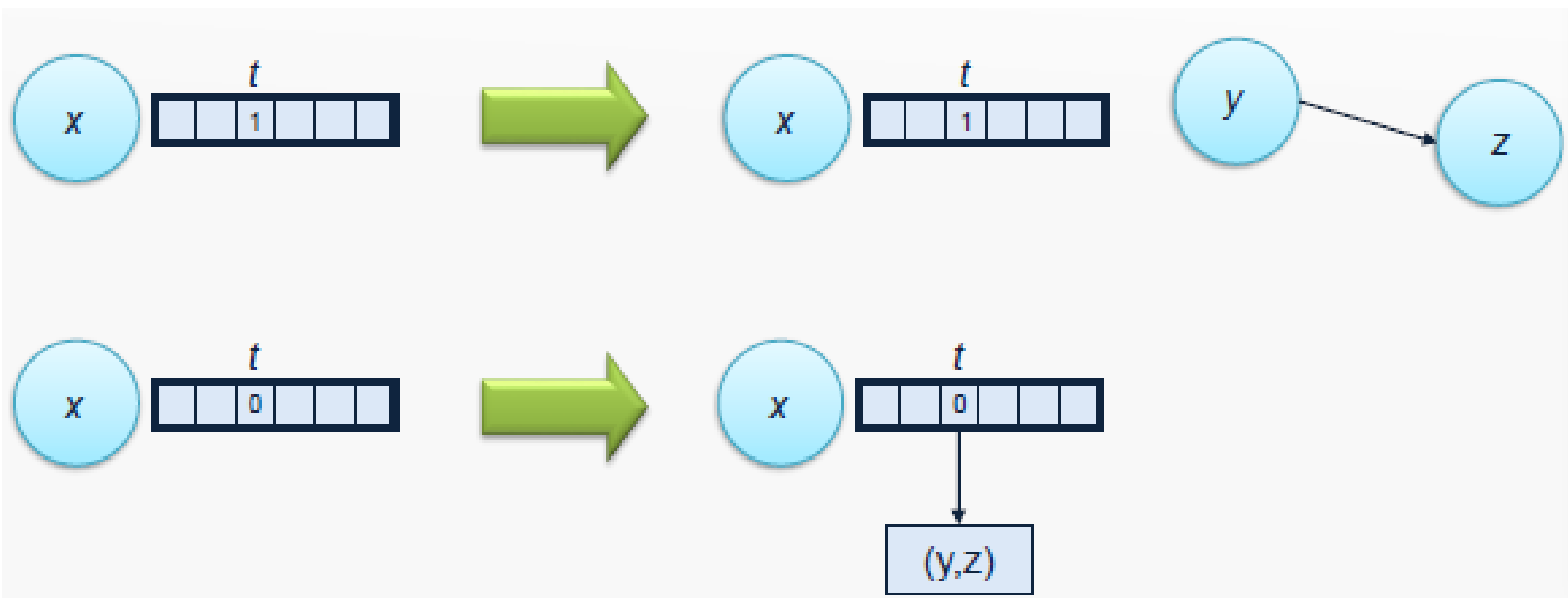
- $t \in x$
- $t \in x \Rightarrow y \subseteq z$



Adding Constraints (2/2)

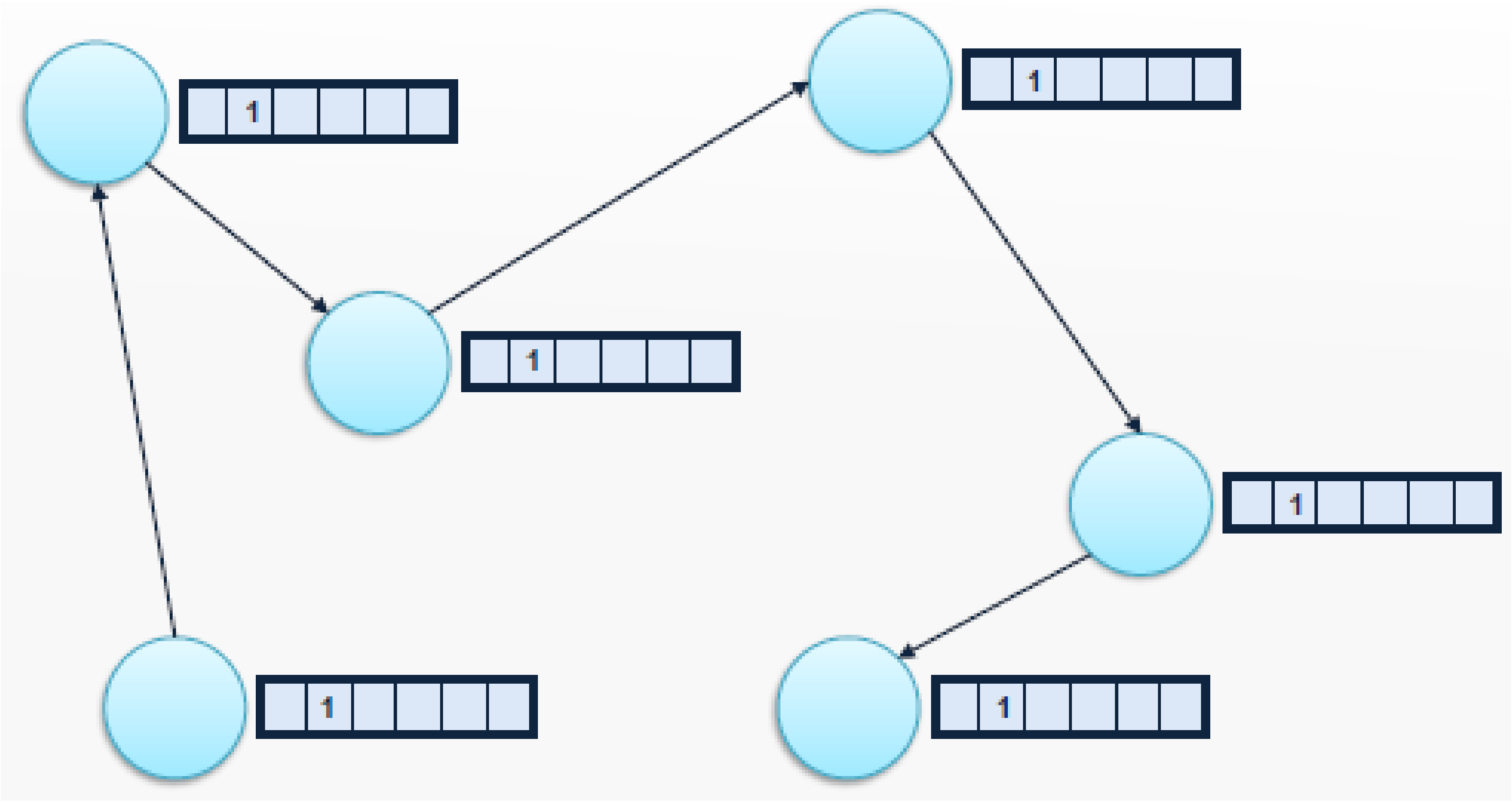
- Constraints of the form $t \in x \Rightarrow y \subseteq z$:
  - test if the bit corresponding to t is 1
  - if so, add the DAG edge from y to z
  - otherwise, add(y,z) to the list of pairs for t

- $t \in x$
- $t \in x \Rightarrow y \subseteq z$



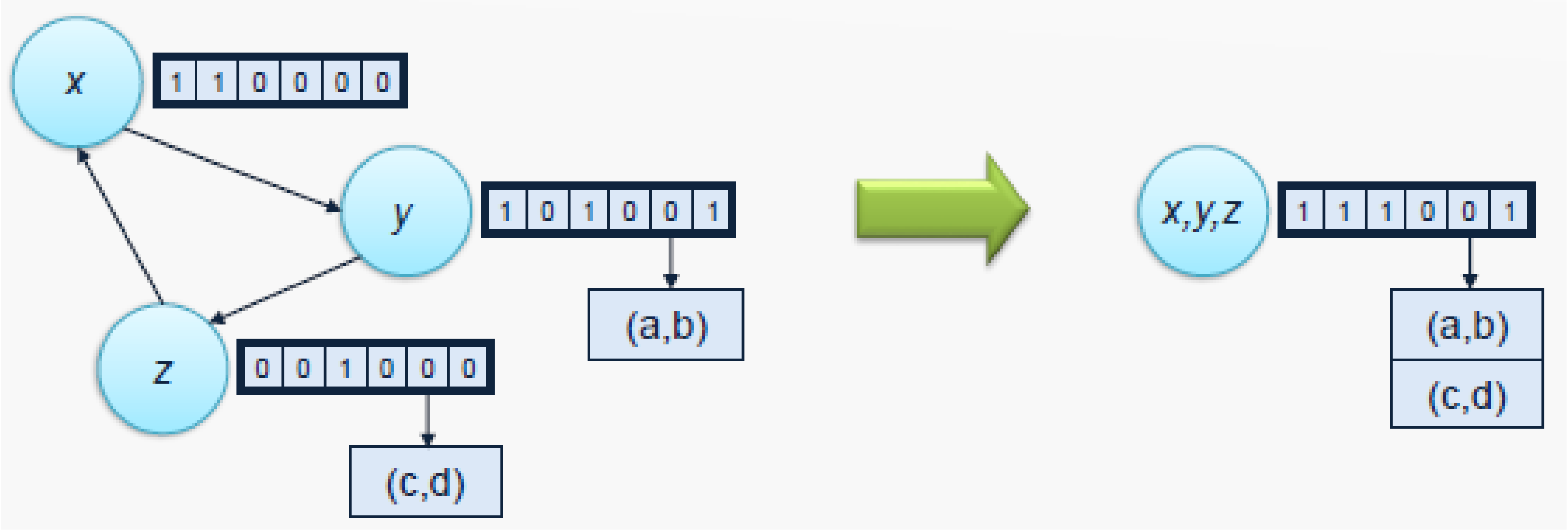
Propagate Bitvectors

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



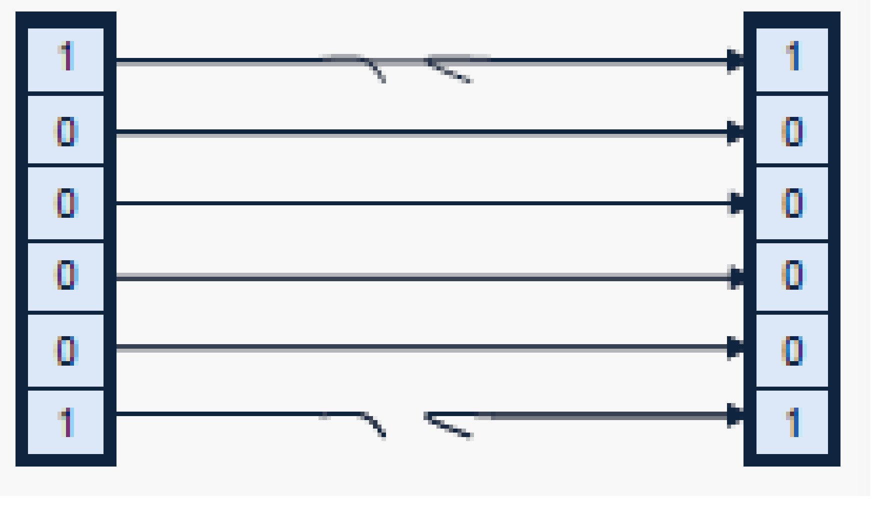
Collapse Cycles

- 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



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(1/2)

- 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 TIP with first-class functions
- Control flow analysis for the λ-calculus
- The cubic framework
- **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?

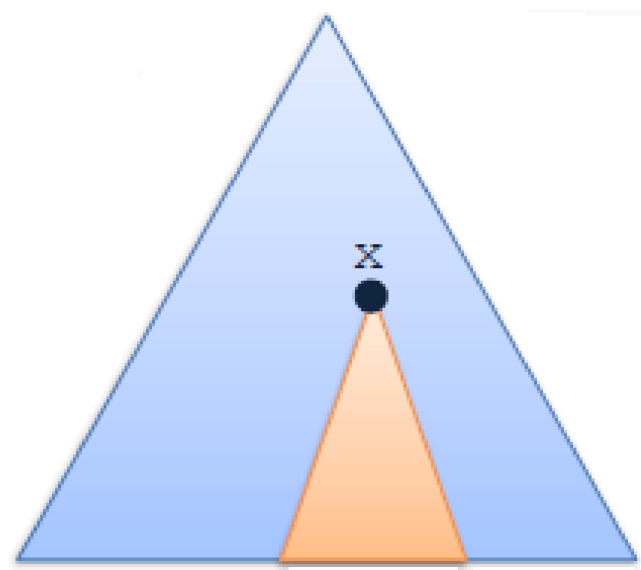
- Full CFA is a possibility...
- But the extra structure allows simpler solutions

Simple CFA for OO (2/3)

- Simplest solution:
  - Select all methods named m with three arguments

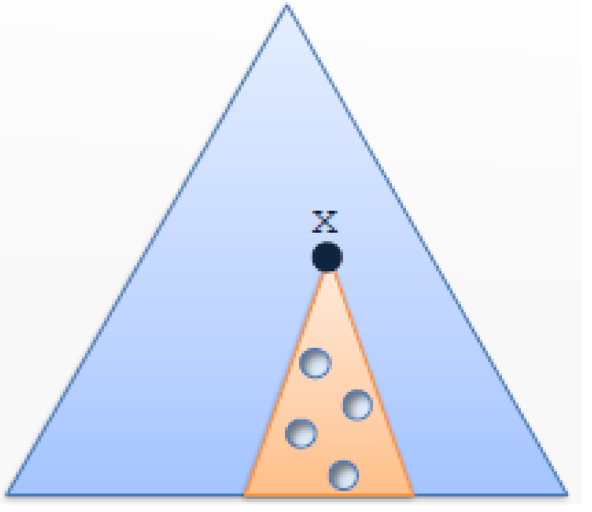
- Class Hierarchy Analysis(CHA):
  - Consider only the part of the class hierarchy rooted by the declared type of x

```
Collection<T> c = ...
c.add(e)
```



Simple CFA for OO (3/3)

- Rapid Type Analysis (RTA):
  - Restrict to those classes that are actually used in the program in **new** expressions
  - Start from **main**, iteratively find reachable methods



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