Generation of Inductive Assertion Network using Interprocedural Data Flow Analysis

Anna Zaks

March 21, 2006

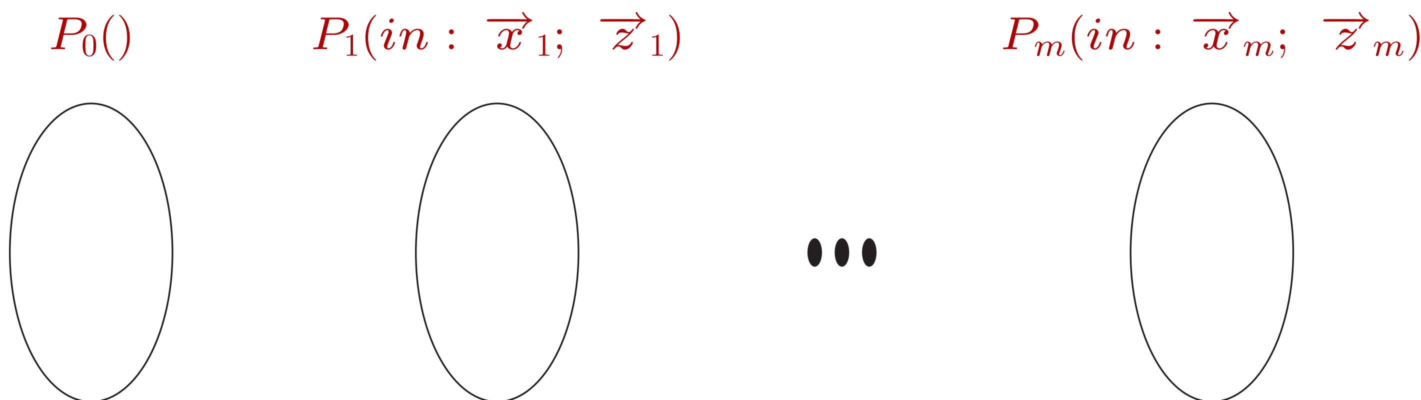
Joint work with: Amir Pnueli

Introduction

- This is a preliminary result.
- Interprocedural Translation Validation as motivation.
- Constant propogation is one of the central optimizations.
- Use source invariants in order to strengthen our proof rules.
- Introduce an existing method for obtaining the precise solution for the constant propagation problem.
- Obtain the source invariants using the solution.

Transition Graphs to Model Programs with Procedures

An application $\mathcal{A}$ consists of $m+1$ modules, where P_0 represents the main procedure:



The variables of each module P_i are partitioned into $\overrightarrow{y} = (\overrightarrow{x}; \overrightarrow{z}; \overrightarrow{w})$, where:

- $\overrightarrow{x}$ are the input parameters passed by value;
- $\overrightarrow{z}$ are the input parameters passed by reference;
- $\overrightarrow{w}$ denotes local(working) variables.

Instructions

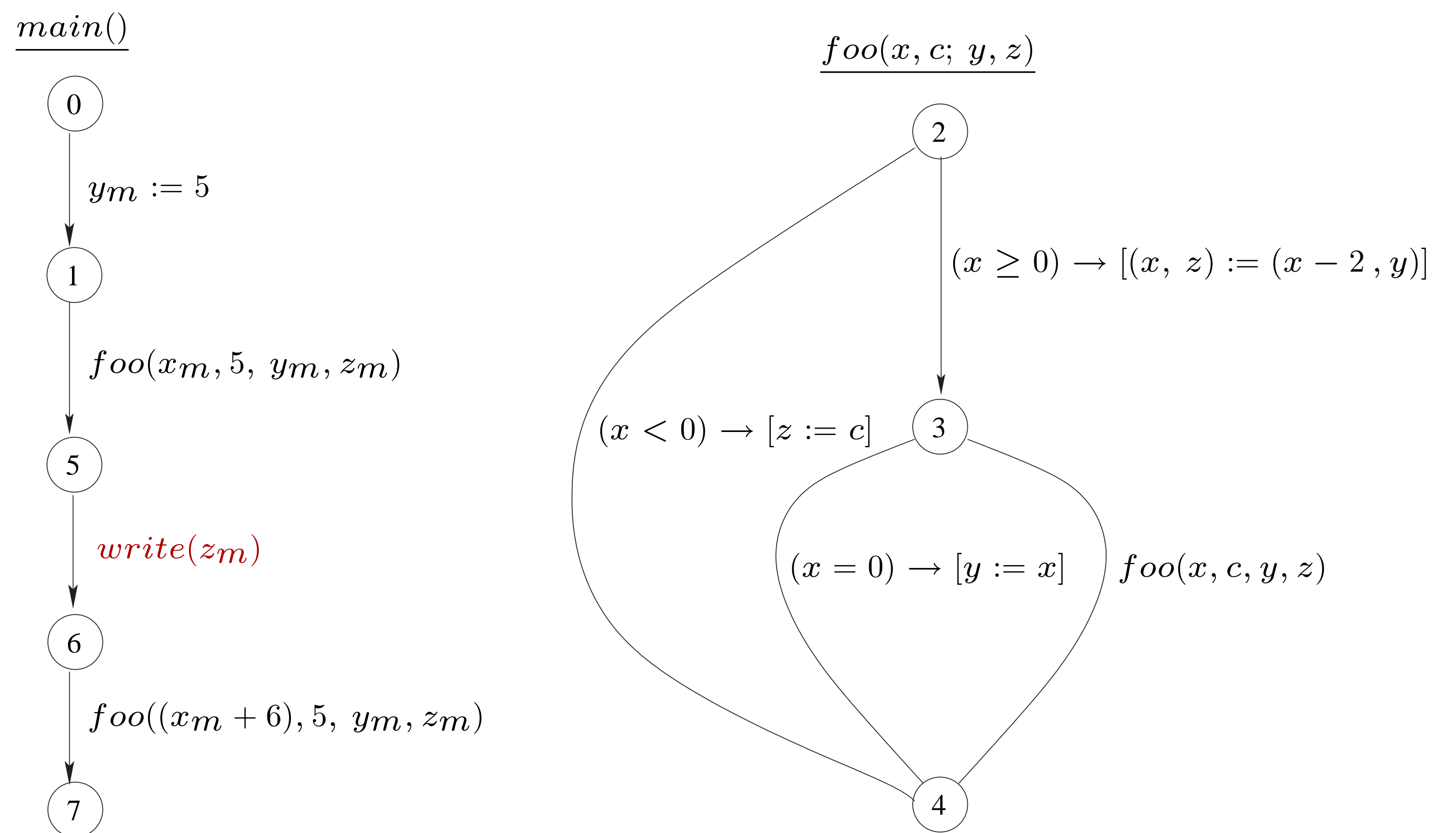
Nodes of the graph are connected by directed edges labeled by instructions:

- A Guarded assignment is an instruction of the form $c \rightarrow [\overrightarrow{u} := \exp(\overrightarrow{y})]$, where c is a boolean expression over $\overrightarrow{y}$, where $\overrightarrow{u} \subseteq \overrightarrow{y}$.
- Read and write instructions are denoted by $read(\overrightarrow{u})$ and $write(\overrightarrow{u})$.
- Procedure call instruction $f(\exp(\overrightarrow{y}), \overrightarrow{u})$ denotes a call to module $f(in : \overrightarrow{x}_f; \overrightarrow{z}_f)$, passing input parameters $\exp(\overrightarrow{y})$ and $\overrightarrow{u}$.

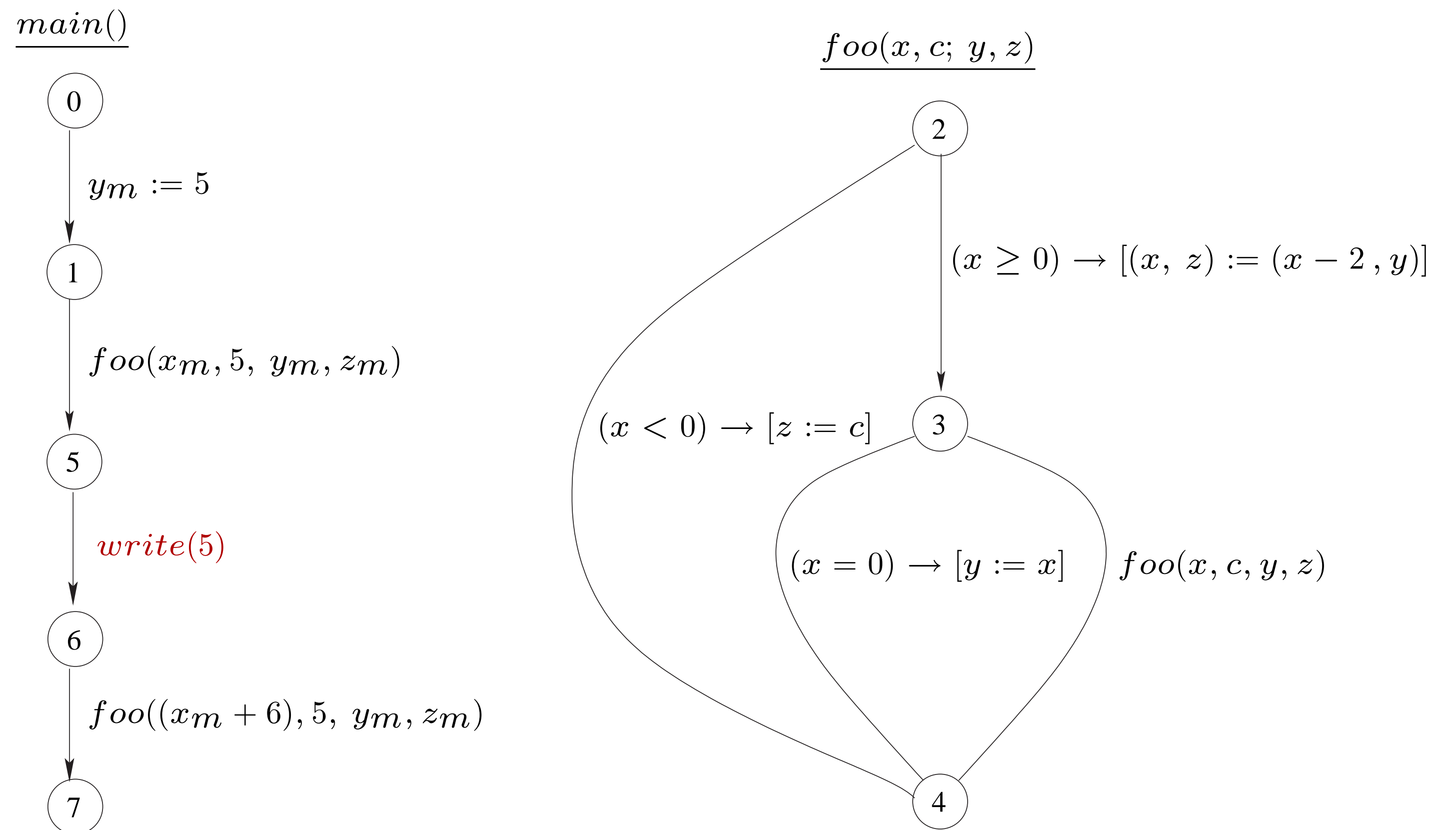
From Programs to Transition Graphs

- We choose the set of cut-points of a procedure P_k to be the set of nodes for the corresponding transition graph.
- If there is a path π from cut-point i to cut-point j, which does not pass through any other cut-point, we add edge (i, j) to the graph and label it by the instruction that summarizes the effect of executing the path π.
- The global variables are efficiently modeled using input parameters.
- The representation of functions is straightforward.

Example of Constant Propagation: Source



Example of Constant Propagation: Target



Assertion Network

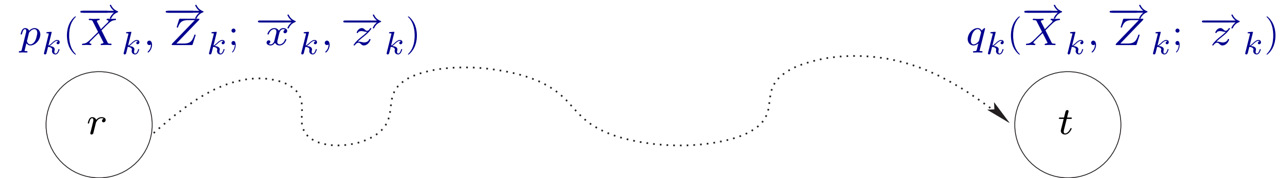
Let fictitious variables $\overrightarrow{X}$ and $\overrightarrow{Z}$ represent the values of the input variables $\overrightarrow{x}$ and $\overrightarrow{z}$ at the procedure entry.

An assertion network $\mathcal{N} = \{\varphi_0, \ldots, \varphi_n\}$ associates an assertion φ_l with each program location l:

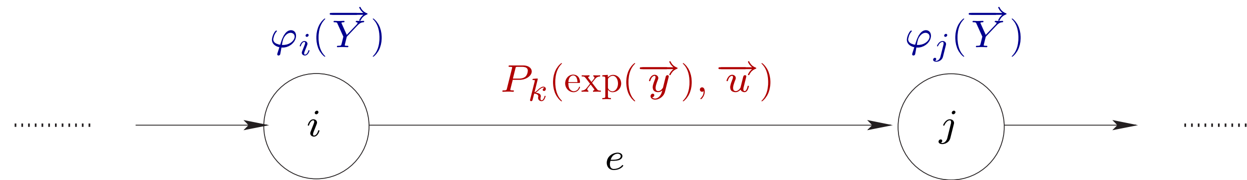
- For each module P_k, the input predicate associated with the procedure entry location is denoted by $p_k(\overrightarrow{X}, \overrightarrow{Z};\ \overrightarrow{x}, \overrightarrow{z})$.
- The assertion associated with the exit location is denoted by $q_k(\overrightarrow{X}, \overrightarrow{Z};\ \overrightarrow{z})$. The output predicate q_k is the procedure summary: it specifies the relation between the input and output values.
- The assertions at all other locations of the procedure $\varphi_l(\overrightarrow{Y})$ may depend on any of the variables.

Verification Conditions: Procedure Call

$\underline{P_k(in:\ \overrightarrow{x}_k;\ \overrightarrow{z}_k)}$



$\underline{P(in:\ \overrightarrow{x};\ \overrightarrow{z})}$



We associate the following two conditions with a procedure call $P_k(\exp(\overrightarrow{y}), \overrightarrow{u})$:

$$\mathcal{VC}_{call}: \quad \varphi_i(\overrightarrow{Y}) \rightarrow p_k(\exp(\overrightarrow{y}), \overrightarrow{u};\ \exp(\overrightarrow{y}), \overrightarrow{u})$$

$$\mathcal{VC}_{return}: \quad \varphi_i(\overrightarrow{Y}) \wedge q_k(\exp(\overrightarrow{y}), \overrightarrow{u};\ \overrightarrow{z}_k) \rightarrow \varphi_j(\overrightarrow{Y})[\overrightarrow{u} \mapsto \overrightarrow{z}_k]$$

Inductive Assertion Network

- An assertion network $\mathcal{N}$ for a program $\mathcal{A}$ is said to be inductive if all the verification conditions for all edges in $\mathcal{A}$ are valid.
- Every inductive network is invariant.

Our goal is to construct an inductive assertion network for the source program that would be strong enough to prove translation in presence of context-sensitive constant propagation.

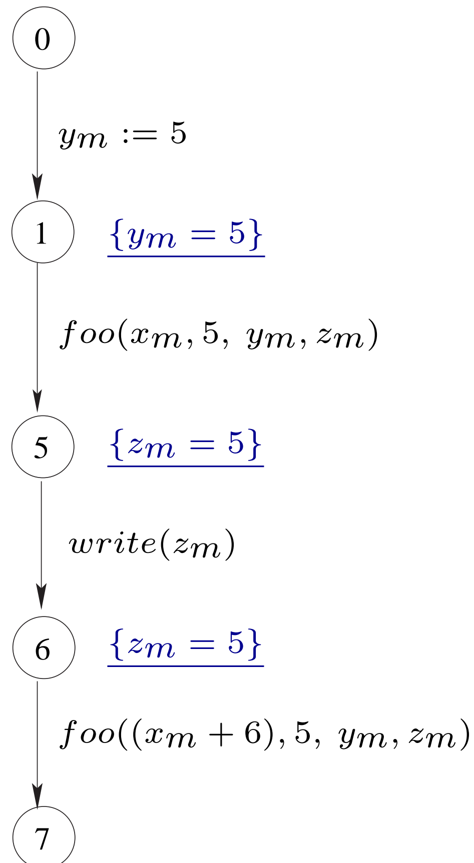
Since the interprocedural constant propagation is not a trivial problem, we compute the inductive network utilizing the existing dataflow analysis.

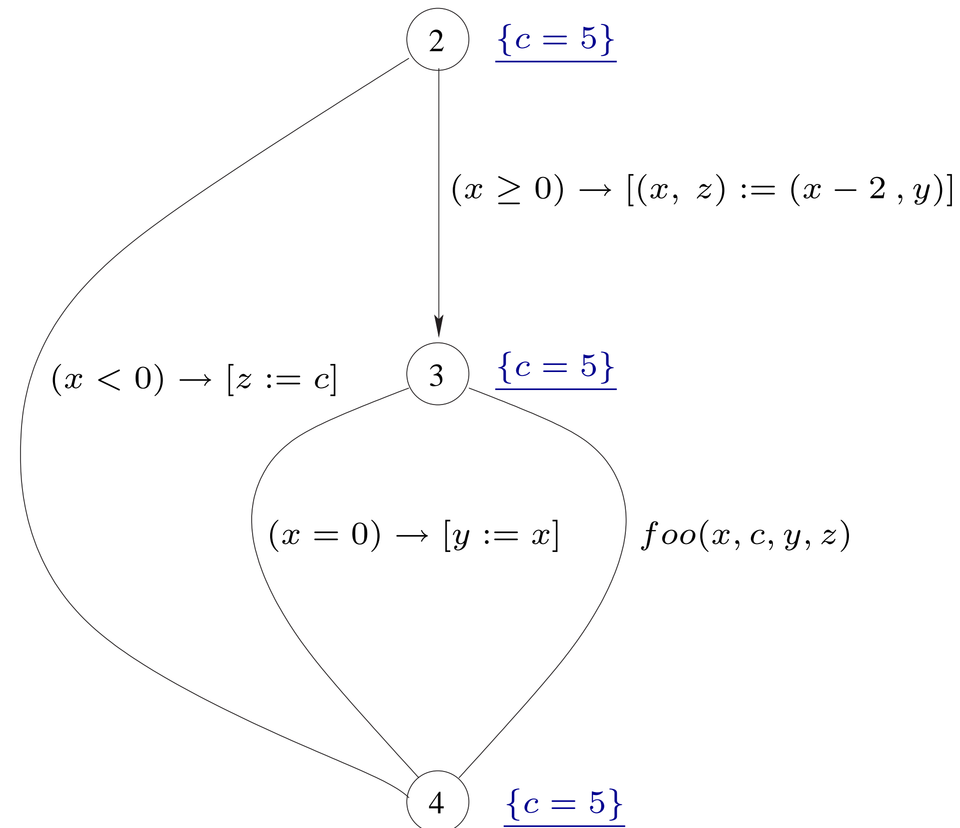
First Attempt

On the first thought, any precise solution to the interprocedural constant propagation problem should suffice: φ_l should be extended with conjunct $x = 5$ if x always evaluates to constant 5 at at location l.

However, the resulting assertion network $\mathcal{N}$ may not be inductive.

Back to our Example





Solution

As our interprocedural dataflow analysis algorithm, we are going to use the one presented in:

- Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation by M. Sagiv, T. Reps, and S. Horwitz.
- Precise Interprocedural Dataflow Analysis via Graph Reachability by M. Sagiv, T. Reps, and S. Horwitz.
- Two Approaches to Interprocedural Dataflow Analysis (Functional Approach) by M. Sharir and A. Pnueli.

The interprocedural dataflow analysis algorithm not only provides a solution to the problem, but also finds a fixpoint for the corresponding set of dataflow equations. Intuitively, we a going to use the information about the fixpoint itself to strengthen our network so it would be inductive.

Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation

- Solves copy constant propagation and linear constant propagation (both distributive problems);
- Produces precise(”meet-over-all-valid-paths” solution) results for both recursive and non-recursive programs;
- The dynamic-programming algorithm solves these constant propagation problems in polynomial time $O(E * MaxVisible^3)$;
- Uses a compact representation of distributive dataflow functions (”environment transformers”).

Distributive Constant Propagation Problems

- In copy constant propagation, a variable x is discovered to be 10 only if
  - $x \ := \ 10$
  - $y \ := \ 10; \ \ x \ := \ y$
- In linear constant propagation, a variable x is discovered to be 10 only if
  - $x \ := \ 10$
  - $y \ := \ 3; \ \ x \ := \ 2 * y \ + \ 4$

Environment Transformer

- Let V be a finite set of program variables. Let $L = Z_{\perp}^{\top}$.
- The set $Env(V, L)$ of environments is the set of functions from V to L.
- A mapping $T : Env(V, L) \mapsto Env(V, L)$ is called an environment transformer.

For an environment $env \in Env(V, L)$, if $env(v) \in Z$ then the variable v has a known constant value in the environment env; the value $\perp$ denotes non constant, and $\top$ denotes an unknown value.

A dataflow problem is specified by annotating each edge of the supergraph of $\mathcal{A}$ with an environment transformer. We are interested in meet over all valid paths solution.

Environment Transformers for Constant Propagation Problems

- Linear constant propagation:

operation	$T : Env(V, L) \mapsto Env(V, L)$
$x := c$	$\lambda env.env[x \rightarrow c]$
$x := c_1 * y + c_2$	$\lambda env.env[x \rightarrow c_1 * env(y) + c_2]$
$x := y + z$	$\lambda env.env[x \rightarrow \bot]$

- The last transformer is a safe approximation; the exact transformer $\lambda env.env[x \rightarrow env(y) + env(z)]$ cannot be used in this framework because it is not distributive.
  - Consider two environments: $env_1 : [y \rightarrow 3; z \rightarrow 2]$ and $env_2 : [y \rightarrow 4; z \rightarrow 1]$
  - Distributivity: $T(env_1 \sqcap env_2) = T(env_1) \sqcap T(env_2)$
  - $T_{x+y}(env_1 \sqcap env_2) = T_{x+y}([y \rightarrow \bot; z \rightarrow \bot]) = [x \rightarrow \bot + \bot] = [x \rightarrow \bot]$
  - $T_{x+y}(env_1) \sqcap T_{x+y}(env_2) = [x \rightarrow 3 + 2] \sqcap [x \rightarrow 4 + 1] = [x \rightarrow 5]$

Pointwise Representation of Environment Transformers

Every distributive transformer $T : Env(V, L) \mapsto Env(V, L)$ can be represented using a set of functions:

$$F^T = \{f_{v,v'} \mid v, v' \in V \cup \{\Lambda\}\}, \text{ each of type } L \mapsto L.$$

- Function $f_{v,v'}$ captures the effect that the value of variable v in the argument environment has on the value of v' in the result environment; if v' does not depend on v, then $f_{v,v'} = \lambda.l.\top$.
- Function $f_{\Lambda,v'}$ is used to represent the effects of on the variable v' that are independent of the argument environment.

For any symbol v', the value $T(env)(v')$ can be determined by taking the meet of the values of $| V | +1$ individual function applications:

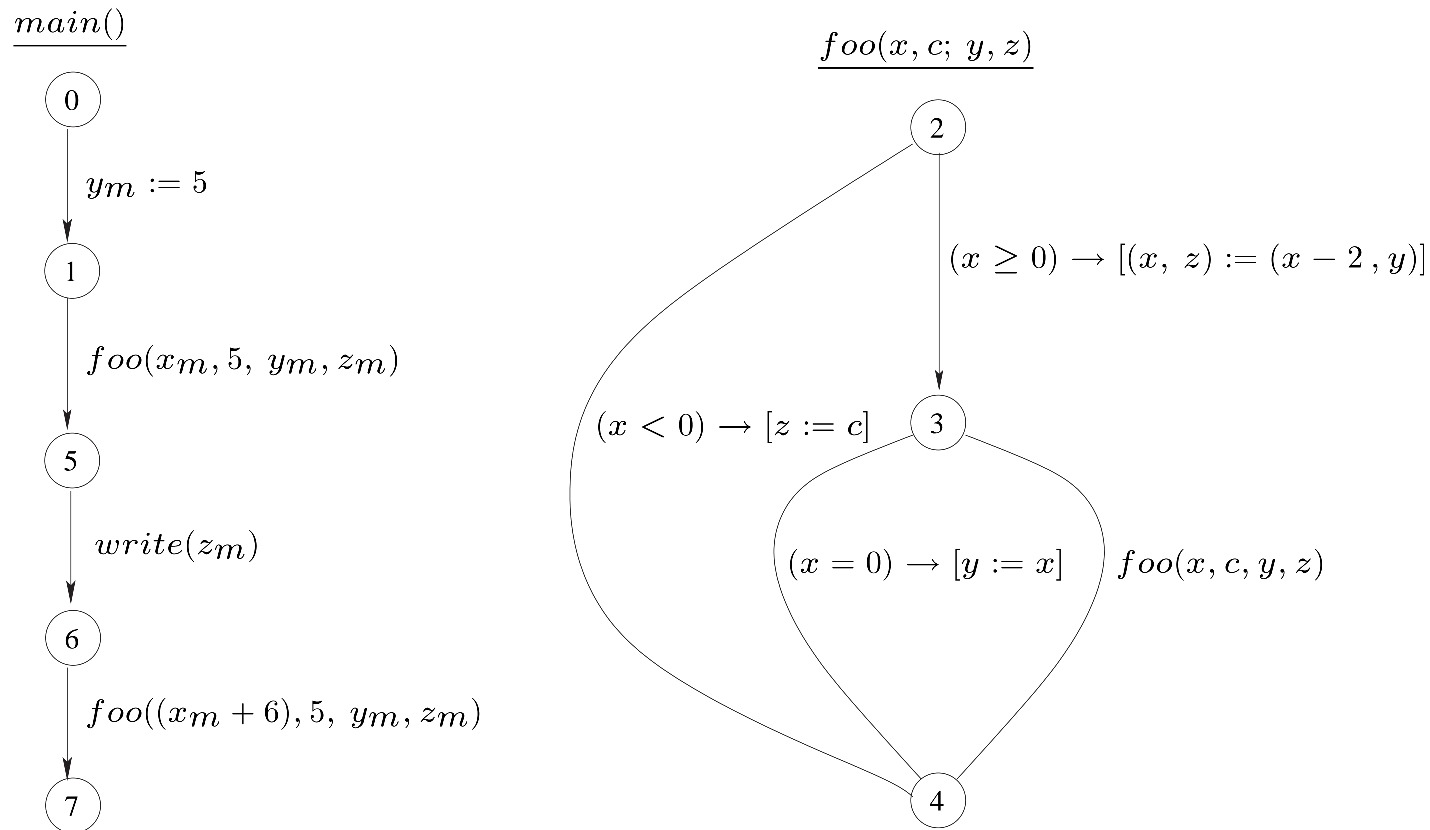
$$T(env)(v') = f_{\Lambda,v'} \sqcap (\sqcap_{v \in V} f_{v,v'}((env)(v))).$$

A Dynamic Programming Algorithm

Two phase algorithm:

- Compute the dataflow solution at the nodes of a procedure as a function of the initial values at the procedure entry - path functions.
- Compute the dataflow values at every point using the path functions.

Phase One: environment transformers



Below is the list of environment transformers computed for procedure foo. We omit all the functions that evaluate to top $f_{(v,v')} = \lambda l.\top$.

$$F_{(2,2)} = \{ f_{x,x} = \lambda l.l,\ f_{c,c} = \lambda l.l,\ f_{y,y} = \lambda l.l,\ f_{z,z} = \lambda l.l \}$$
$$F_{(2,3)} = \{ f_{c,c} = \lambda l.l,\ f_{y,y} = \lambda l.l,\ f_{y,z} = \lambda l.l \}$$
$$F_{(2,4)} = \{ f_{c,c} = \lambda l.l,\ f_{c,z} = \lambda l.l,\ f_{y,z} = \lambda l.l \}$$

Computation of Invariants using the Env Transformers

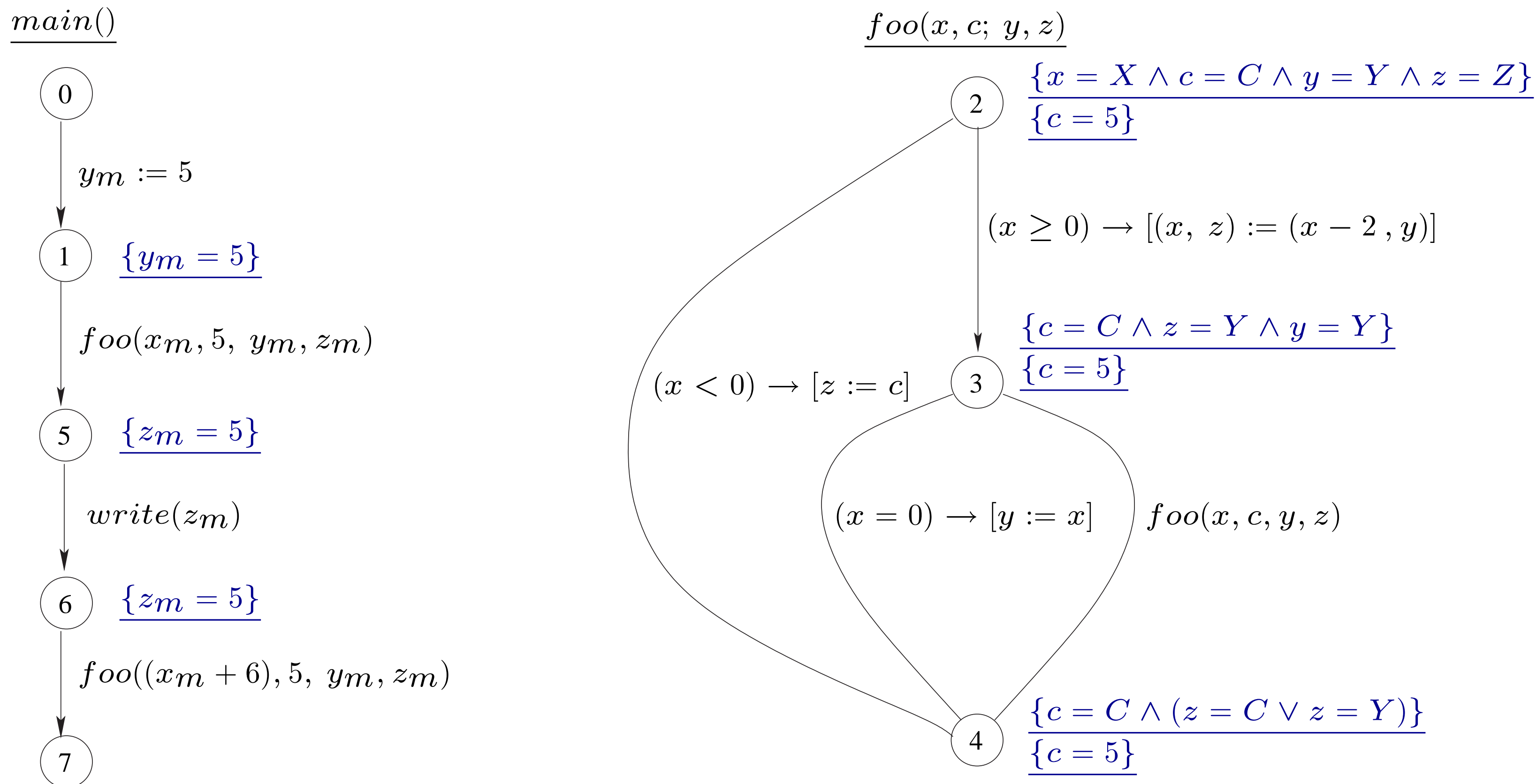
Given all dataflow facts and the transformer represented by $F_{(i,j)}$, we follow the following rules to compute an invariant φ_l at location l of P_k:

- We ignore all functions of the form $f_{(v,v')} = \lambda l.\top$.
- For each variable v' that is not set to $\perp$ by $f_{(\Lambda,v')} \in F_{(r_k,l)}$ we add the following conjunct to φ_l:

  $$\bigvee_{f_{v,v'} \in F_{(r_k,l)}} v' = f_{v,v'}(V)$$

  Note that we use disjunction to model the effect of the meet operator.
- We also add the conjunct $x = const$ if x was determined to evaluate to constant $const$ at at location l. We need this addition since $T_{(r_k,l)}$ does not propagate the the information from the callers.

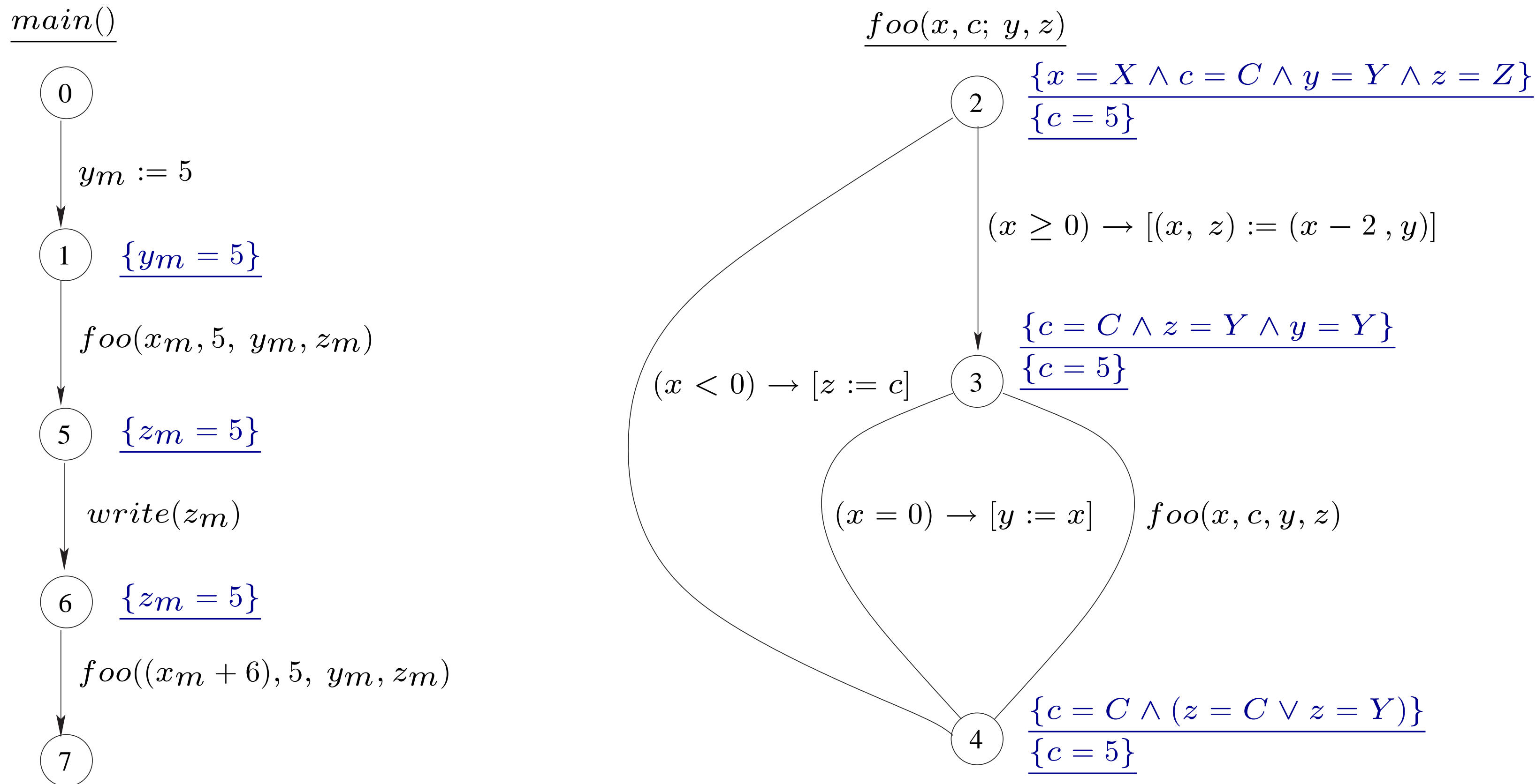
Resulting Inductive Assertion Network



The list of environment transformers computed for procedure foo:

$$
\begin{aligned}
F_{(2,2)} &= \quad \{\, f_{x,x} = \lambda l.l,\ f_{c,c} = \lambda l.l,\ f_{y,y} = \lambda l.l,\ f_{z,z} = \lambda l.l \,\} \\
F_{(2,3)} &= \quad \{\, f_{c,c} = \lambda l.l,\ f_{y,y} = \lambda l.l,\ f_{y,z} = \lambda l.l \,\} \\
F_{(2,4)} &= \quad \{\, f_{c,c} = \lambda l.l,\ f_{c,z} = \lambda l.l,\ f_{y,z} = \lambda l.l \,\}
\end{aligned}
$$

Resulting Inductive Assertion Network



For example, let's show that the return verification condition for call edge $(1, 5)$ of our example holds.

$$\begin{array}{llllll} \mathcal{VC}_{ret}: & \varphi_1 & \wedge & \varphi_4[(C, Y) \mapsto (5, y_m)] & \rightarrow & \varphi_5[z_m \mapsto z] & \Leftrightarrow \\ & y_m = 5 & \wedge & c = 5 \wedge (z = 5 \vee z = y_m) \wedge c = 5 & \rightarrow & z = 5 & \end{array}$$

Questions?