# Validation of Interprocedural Optimizations

Amir Pnueli, Anna Zaks

New York University

COCV'08, Budapest, Hungary, April $5^{th}$, 2008

# Translation Validation

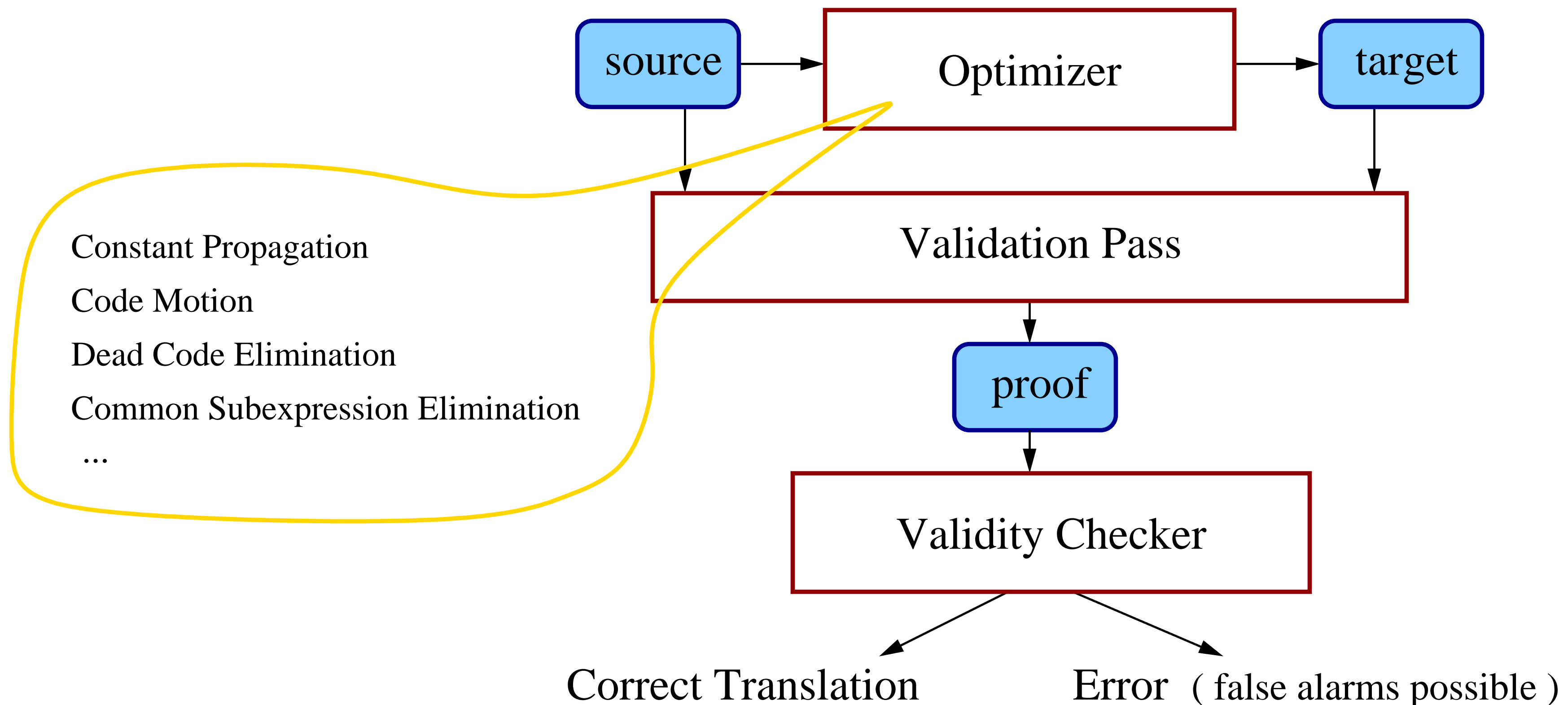
Translation validation frameworks can be categorized by the intended user:

- Compiler writers are interested in methods that lead to creation of a self-certifying compiler. The full control over the compiler allows for powerful and efficient techniques.
  - Blech and Poetzsch-Heffter: A certifying code generation phase, 2007
  - Tristan and Leroy: Formal verification of translation validators: a case study on instruction scheduling optimizations, 2008
- Compiler users may need to work with an existing compiler and require tools that accommodate minimal compiler cooperation.
  - Pnueli, Siegel, Singerman: Translation validation, 1998
  - Necula: Translation validation for an optimizing compiler, 2000
  - Zuck, Pnueli, Fang, Goldberg: VOC: A methodology for the translation validation of optimizing compilers, 2003
  - Rival: Symbolic transfer function-based approaches to certified compilation, 2004

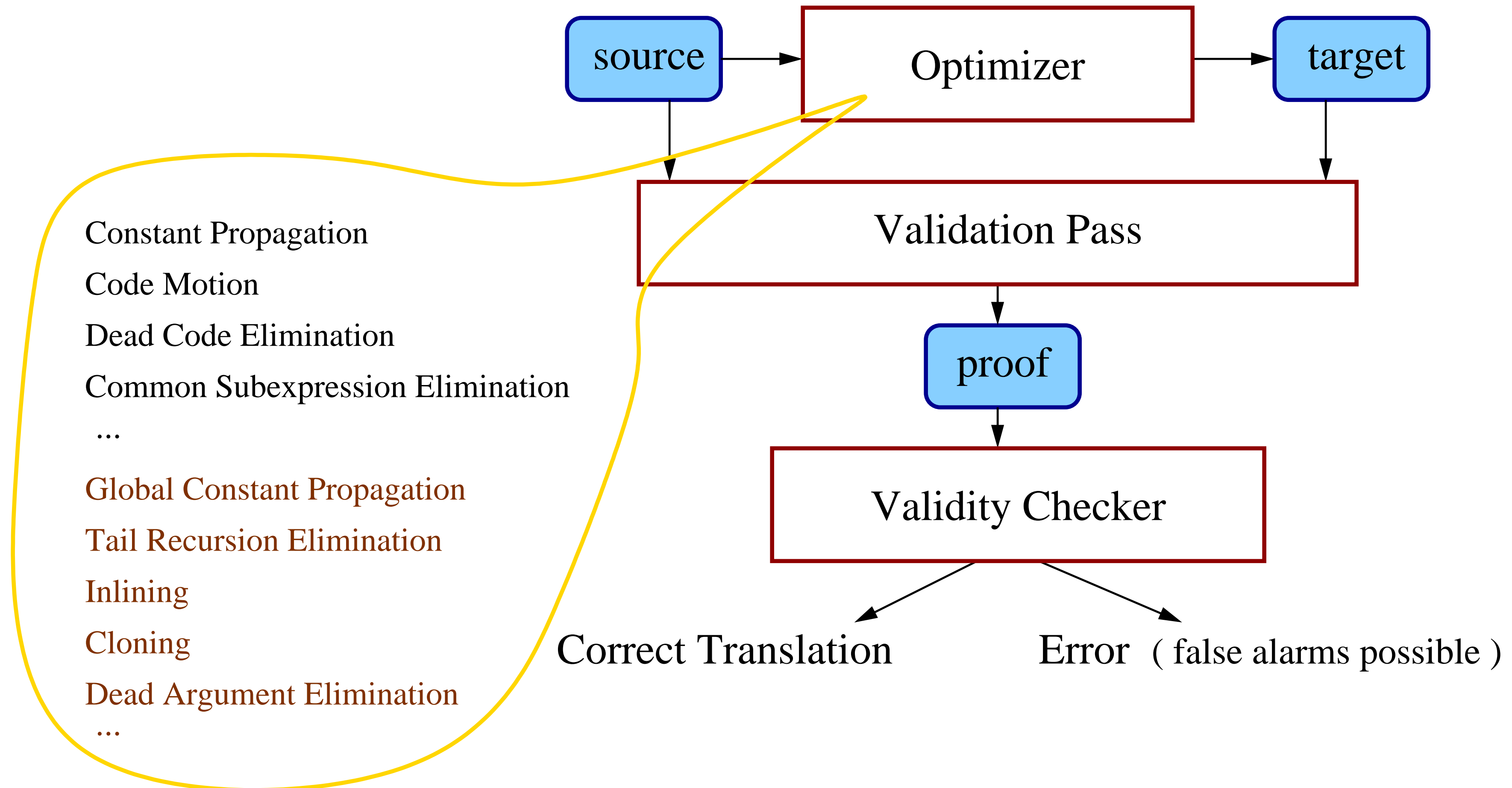
Currently, the existing translation validation approaches do not deal with interprocedural optimizations. Our work is an extension of the VOC framework to deal with procedural programs.

# Overview of VOC

- Each optimized compiler run is followed by a validation pass that generates a proof of translation correctness. The proof is later checked by a third-party validity checker (CVC 3).
- The method does not require a specialized compiler; however, it utilizes compiler debug information to minimize false alarms.



# Our Contribution



# Roadmap

- Transition Graphs as our formal model of programs with procedures.
- The notion of Correct Translation.
- The Validation Algorithm that constructs a proof of Correct Translation in presence of interprocedural optimizations like:
  - global constant propagation,
  - inlining,
  - tail recursion elimination,
  - dead argument elimination,
  - and others.
- Generation of inductive assertion network, required to handle global constant propagation.

# Transition Graphs

Each procedure is represented by a transition graph.

Nodes of the graph $N$ must include all the program cut-points.

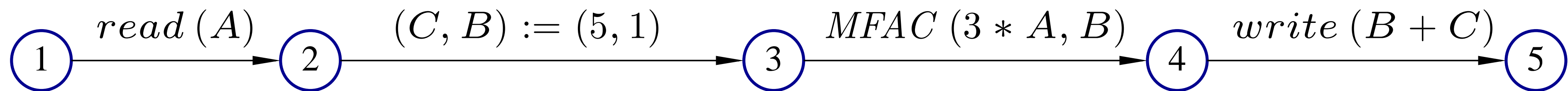
The nodes are connected by directed edges labeled by instructions:

- A Guarded assignment is an instruction of the form $c \rightarrow [\vec{u} := \vec{E}(\vec{y})]$, where $c$ is a boolean expression.
- Read and write instructions are denoted by $read(\vec{u})$ and $write(\vec{u})$.
- Procedure call instruction $f(\vec{E}(\vec{y}); \vec{u})$ denotes a call to procedure $f(\vec{x}_f;\ \&\ \vec{z}_f)$, passing input parameters $\vec{E}(\vec{y})$ by value and output parameters $\vec{u}$ by reference.

# Factorial Example

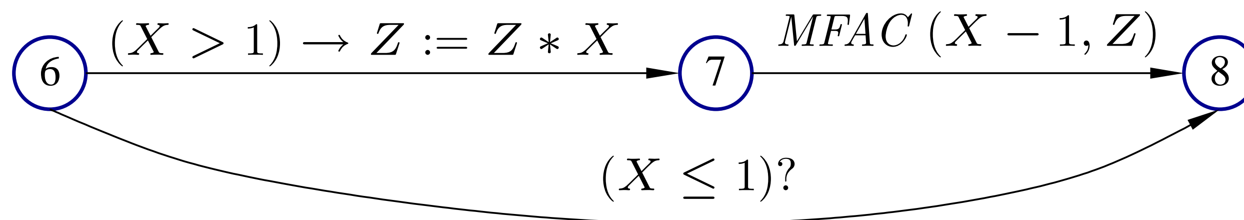
$MAIN()$

{ On input $A$, outputs: $(3 * A)! + 5$ }




$MFAC\,(X; \&Z)$

{ Computes in Z: $Z * X!$ }



# Correct Translation

- An observation of a program $\mathcal{A}$ is obtained from a program computation by setting to $\top$ the transitions and data that do not participate in a read or write instructions.

  For example, below is an observation of a program that reads two numbers and then writes out their product:

$$\top \quad \xrightarrow{read} \quad (5, 22) \quad \xrightarrow{\top} \quad \top \quad \xrightarrow{\top} \quad \top \quad \xrightarrow{\top} \quad \top \quad \xrightarrow{\top} \quad (110) \quad \xrightarrow{write} \quad \top$$

- Observations $o'$ and $o$ are stuttering equivalent if they differ from each other by finite sequences of pairs $\top \xrightarrow{\top}$ or $\xrightarrow{\top} \top$.

  For instance, the observation above is stuttering equivalent to the following:

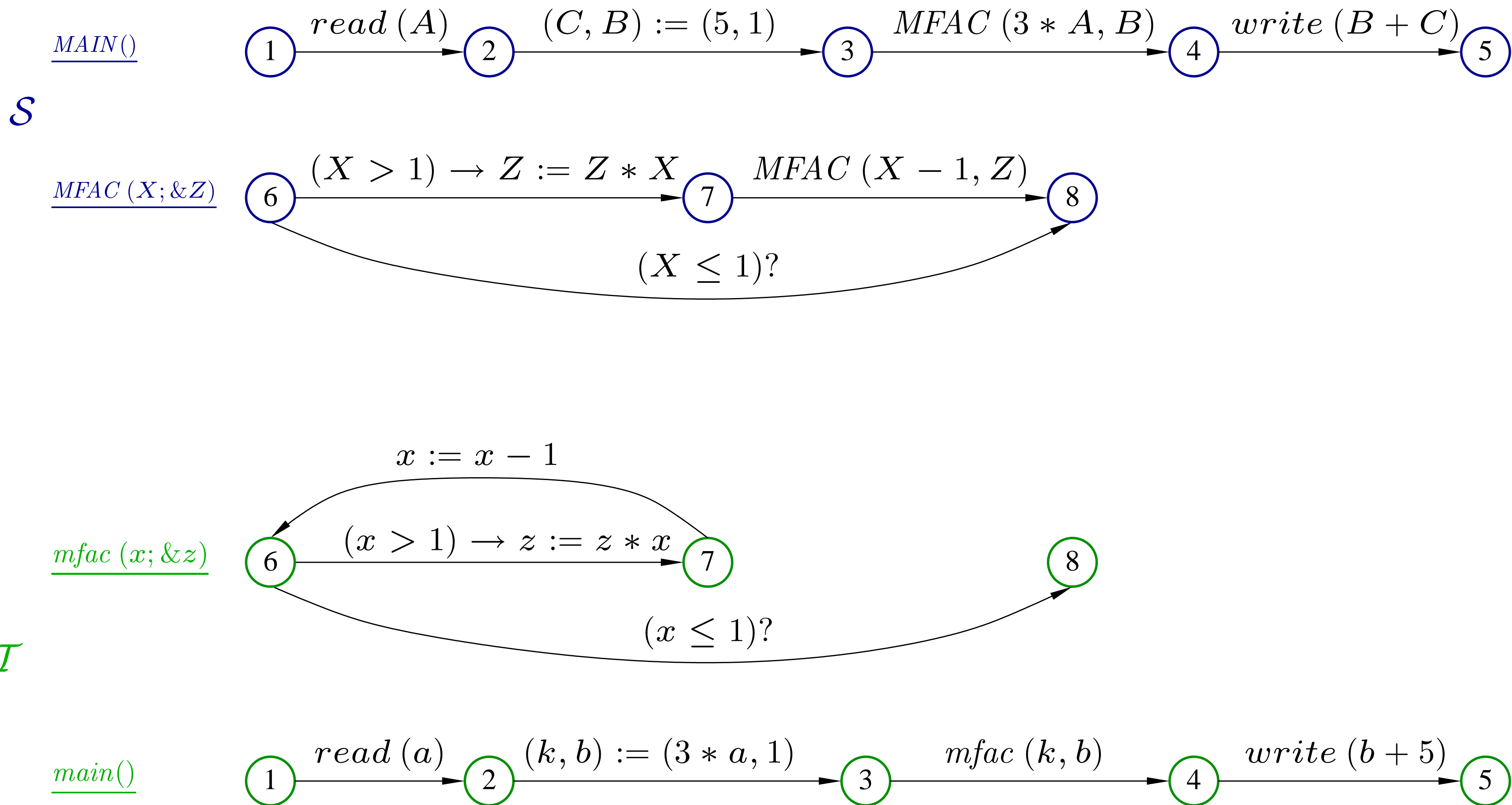
$$\top \quad \xrightarrow{read} \quad (5, 22) \quad \xrightarrow{\top} \quad \top \quad \xrightarrow{\top} \quad (110) \quad \xrightarrow{write} \quad \top$$

- Program $\mathcal{T}$ is a correct translation (refinement) of program $\mathcal{S}$ if, for every observation $o_{\mathcal{T}} \in Obs(\mathcal{T})$, there exists a stuttering equivalent observation $o_{\mathcal{S}} \in Obs(\mathcal{S})$.

# Translation Validation Algorithm

Goal: Given two programs $\mathcal{S}$ and $\mathcal{T}$,
construct a proof showing that the target program $\mathcal{T}$ is a correct translation of the source $\mathcal{S}$.

# Factorial Example



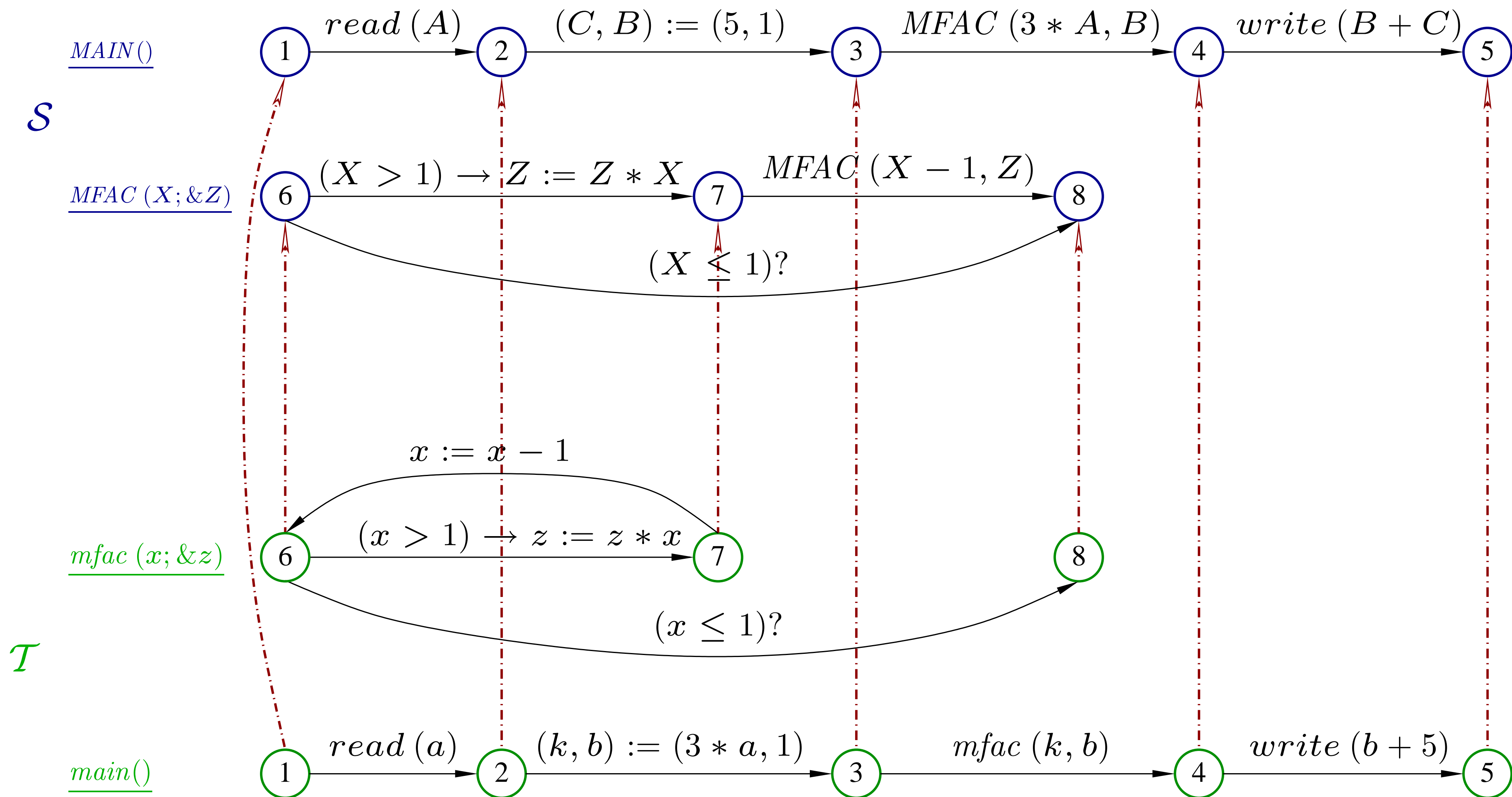
# Translation Validation Algorithm

Goal: Given two programs $\mathcal{S}$ and $\mathcal{T}$,
construct a proof showing that the target program $\mathcal{T}$ is a correct translation of the source $\mathcal{S}$.

1. Establish control mapping $\kappa : N^{\mathcal{T}} \rightarrow N^{\mathcal{S}}$ mapping the target nodes to the source nodes, such that $r$ is the initial location of $\mathcal{T}$ iff $\kappa(r)$ is the initial location of $\mathcal{S}$.
   - The mapping is total.
   - Optimizations such as inlining result in a non-injective (many-to-one) control mapping.
   - $\kappa$ will not be surjective (onto) if dead code elimination removes a loop.

   The compiler debug information is used to establish the correspondence between the source and target nodes.

# Factorial Example: Control Mapping



# Translation Validation Algorithm

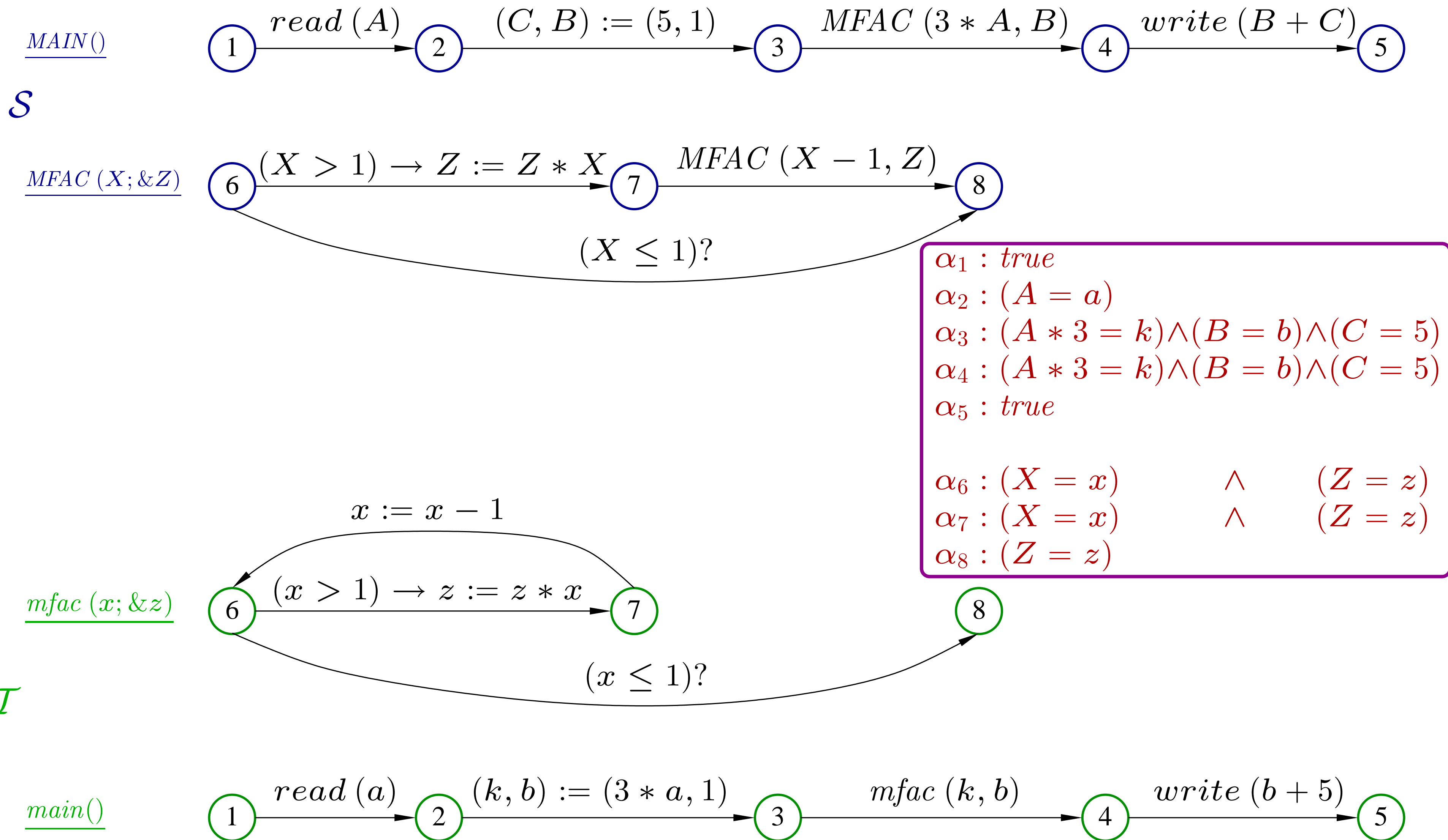
Goal: Given two programs $\mathcal{S}$ and $\mathcal{T}$,
construct a proof showing that the target program $\mathcal{T}$ is a correct translation of the source $\mathcal{S}$.

1. Establish control mapping $\kappa : N^{\mathcal{T}} \rightarrow N^{\mathcal{S}}$ mapping the target nodes to the source nodes.

2. Form data abstraction $\mathcal{D} = \{\alpha_0, \ldots, \alpha_{|N^{\mathcal{T}}|}\}$ by defining each $\alpha_l(V^{\mathcal{S}}; V^{\mathcal{T}})$, for target location $l \in N^{\mathcal{T}}$, as a conjunction of equalities of the form $E(V^{\mathcal{S}}) = E(V^{\mathcal{T}})$. Note that implicitly, $\alpha_l$ is associated with a pair of target and source locations $l$ and $\kappa(l)$. Thus, the data abstraction relates values of target variables to those of source variables.

   Following [VOC 2003], the data abstraction is based on the specialized data flow analysis and the mapping between the source and target variables available from the compiler debug information.

# Factorial Example: Data Abstraction



# Translation Validation Algorithm

Goal: Given two programs $\mathcal{S}$ and $\mathcal{T}$,
construct a proof showing that the target program $\mathcal{T}$ is a correct translation of the source $\mathcal{S}$.

1. Establish control mapping $\kappa : N^{\mathcal{T}} \rightarrow N^{\mathcal{S}}$ mapping the target nodes to the source nodes.

2. Form data abstraction $\mathcal{D} = \{\alpha_0, \ldots, \alpha_{|N^{\mathcal{T}}|}\}$ by defining each $\alpha_l(V^{\mathcal{S}}; V^{\mathcal{T}})$, for target location $l \in N^{\mathcal{T}}$, as a conjunction of equalities of the form $E(V^{\mathcal{S}}) = E(V^{\mathcal{T}})$. The data abstraction must be valid at the initial location of $\mathcal{T}$, $\alpha_r = true$.

3. Generate translation verification conditions and place them in a set $\mathcal{VC}$, which forms an inductive proof of correct translation.

   - Form the initial verification condition: $true \rightarrow \alpha_r$
   - Form translation verification conditions for every edge of the target program.

# I/O Verification Conditions

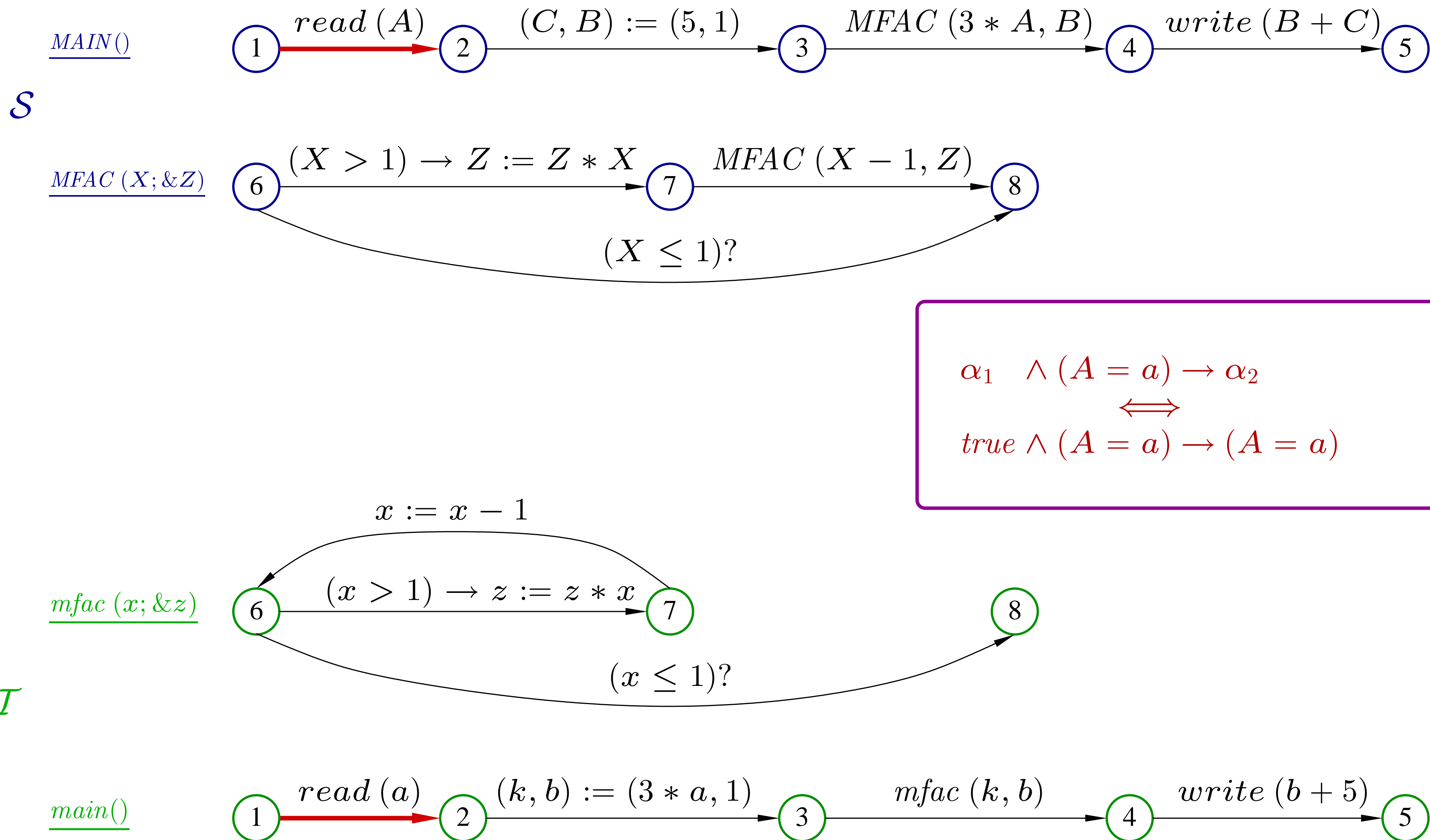
Read: For $e^{\mathcal{T}} = (i, j)$ and $e^{\mathcal{S}} = (\kappa(i), \kappa(j))$, labeled $read(\vec{u}^{\mathcal{T}})$ and $read(\vec{u}^{\mathcal{S}})$:

$$\alpha_i \wedge (\vec{u}^{\mathcal{T}} = \vec{u}^{\mathcal{S}}) \rightarrow \alpha_j$$

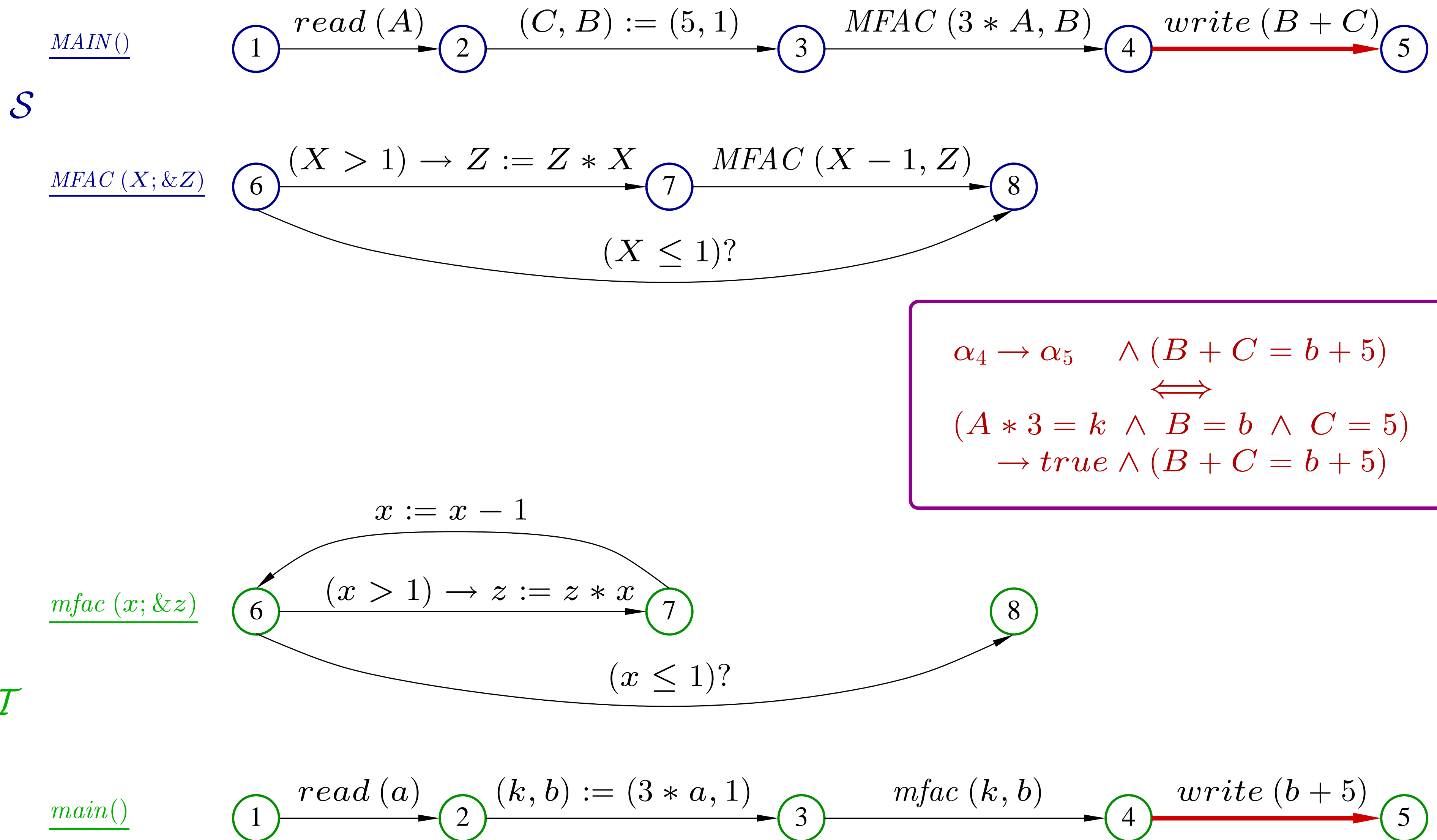
Write: For $e^{\mathcal{T}} = (i, j)$ and $e^{\mathcal{S}} = (\kappa(i), \kappa(j))$, labeled by $write(\vec{E}^{\mathcal{T}})$ and $write(\vec{E}^{\mathcal{S}})$:

$$\alpha_i \rightarrow \alpha_j \wedge (\vec{E}^{\mathcal{T}} = \vec{E}^{\mathcal{S}})$$

# Factorial Example: Read Verification Condition



# Factorial Example: Write Verification Condition



# Assignment Verification Condition

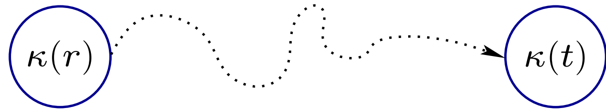
Assignment Verification Condition is

- left as an exercise for the reader.
- very similar to the condition presented in [VOC 2003].

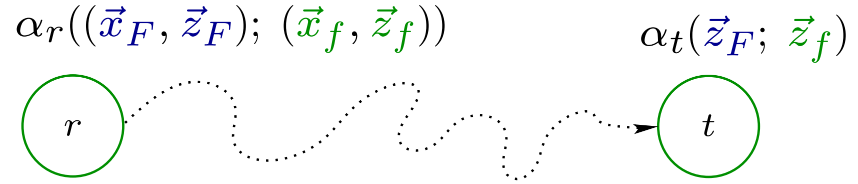
# Call Verification Conditions

Call Verification Conditions are generated, when both $e^{\mathcal{T}}$ and $e^{\mathcal{S}}$ are call edges.

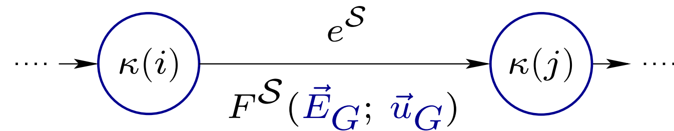
$F^{\mathcal{S}}(\vec{x}_F;\ \&\ \vec{z}_F)$



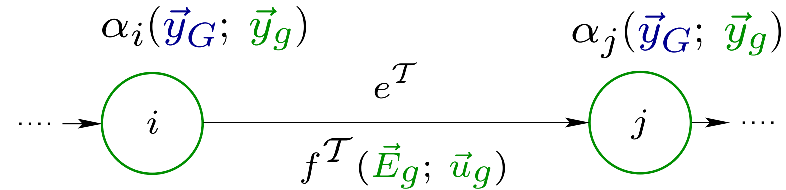

$f^{\mathcal{T}}(\vec{x}_f;\ \&\ \vec{z}_f)$




$G^{\mathcal{S}}(\ldots)$




$g^{\mathcal{T}}(\ldots)$

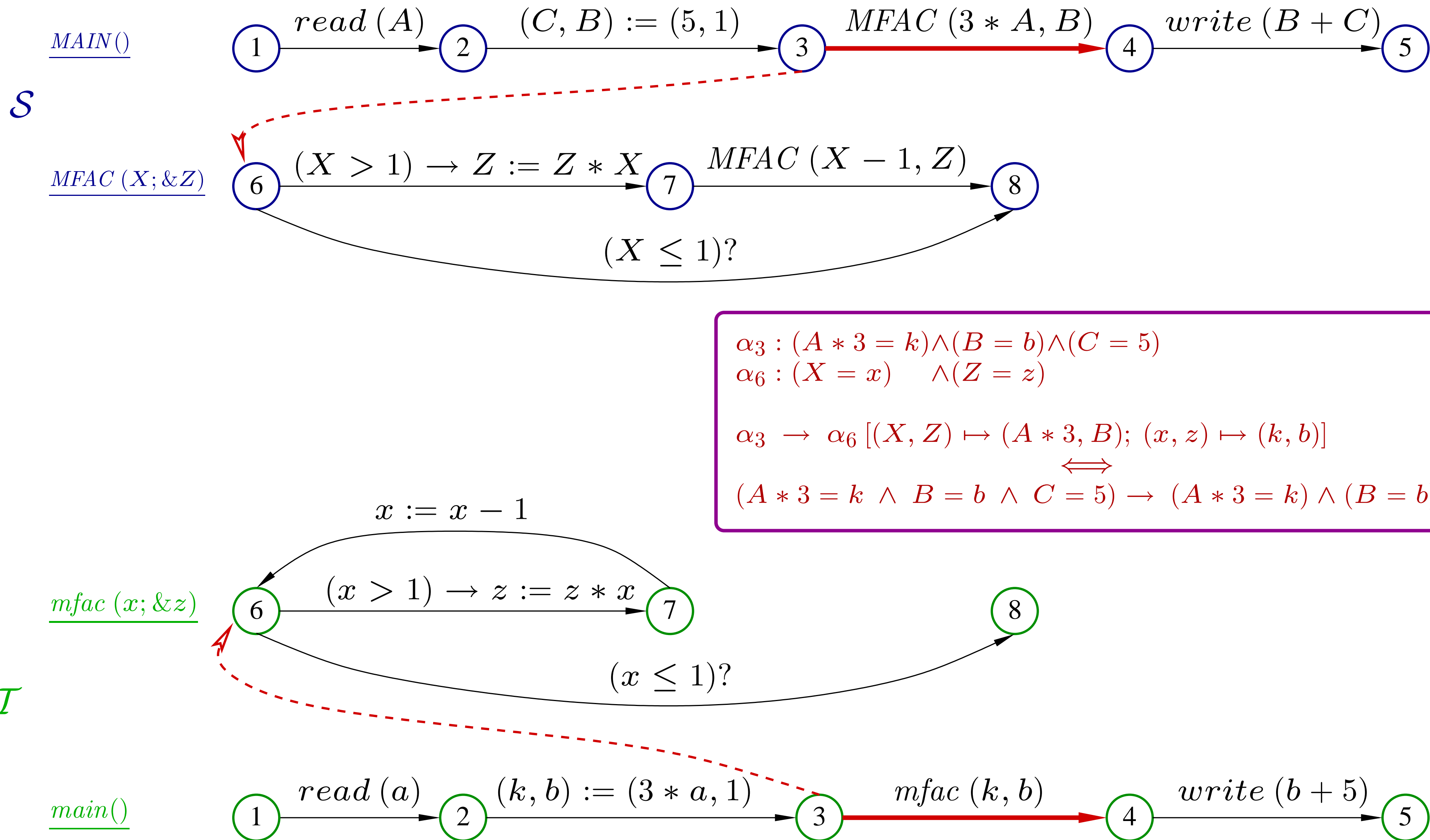



$$\mathcal{VC}_{call}: \quad \alpha_i(\vec{y}_G;\ \vec{y}_g) \quad \rightarrow \quad \alpha_r((\vec{E}_G, \vec{u}_G);\ (\vec{E}_g, \vec{u}_g))$$

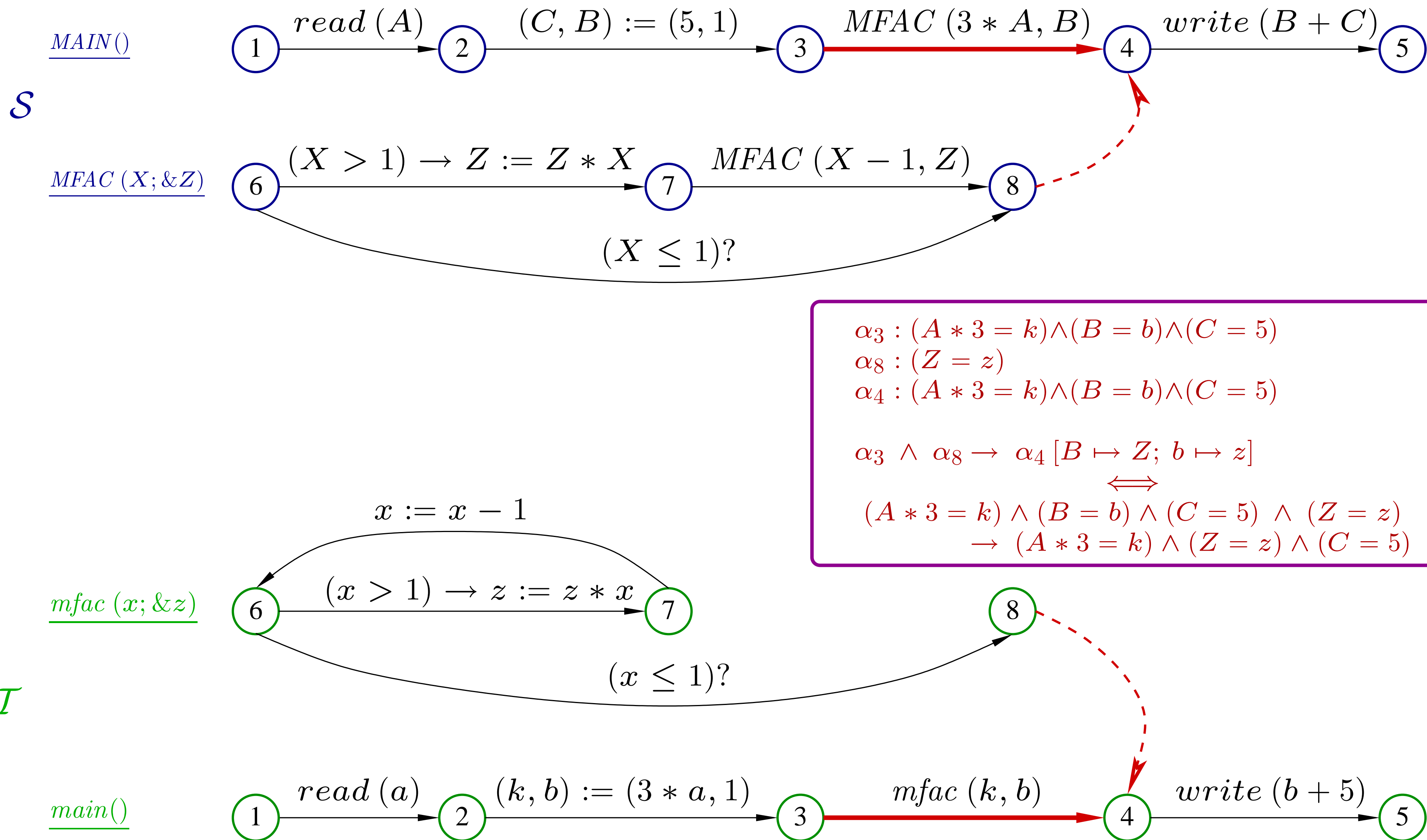
$$\mathcal{VC}_{ret}: \quad \alpha_i(\vec{y}_G;\ \vec{y}_g) \wedge \alpha_t(\vec{z}_F;\ \vec{z}_f) \quad \rightarrow \quad \alpha_j(\vec{y}_G;\ \vec{y}_g)\ [\vec{u}_G \mapsto \vec{z}_F; \vec{u}_g \mapsto \vec{z}_f]$$

where $\alpha_j(\vec{y}_G;\ \vec{y}_g)\ [\vec{u}_G \mapsto \vec{z}_F; \vec{u}_g \mapsto \vec{z}_f]$ is obtained from $\alpha_j(\vec{y}_G; \vec{y}_g)$ by replacing variables in $\vec{u}_G$ by the corresponding variables in $\vec{z}_F$ and variables in $\vec{u}_g$ by the variables in $\vec{z}_f$.

# Factorial Example: $\mathcal{VC}_{call}$



# Factorial Example: $\mathcal{VC}_{return}$




$\alpha_3 : (A * 3 = k) \wedge (B = b) \wedge (C = 5)$
$\alpha_8 : (Z = z)$
$\alpha_4 : (A * 3 = k) \wedge (B = b) \wedge (C = 5)$

$$\alpha_3 \wedge \alpha_8 \rightarrow \alpha_4 [B \mapsto Z;\ b \mapsto z]$$
$$\Longleftrightarrow$$
$$(A * 3 = k) \wedge (B = b) \wedge (C = 5) \wedge (Z = z) \rightarrow (A * 3 = k) \wedge (Z = z) \wedge (C = 5)$$

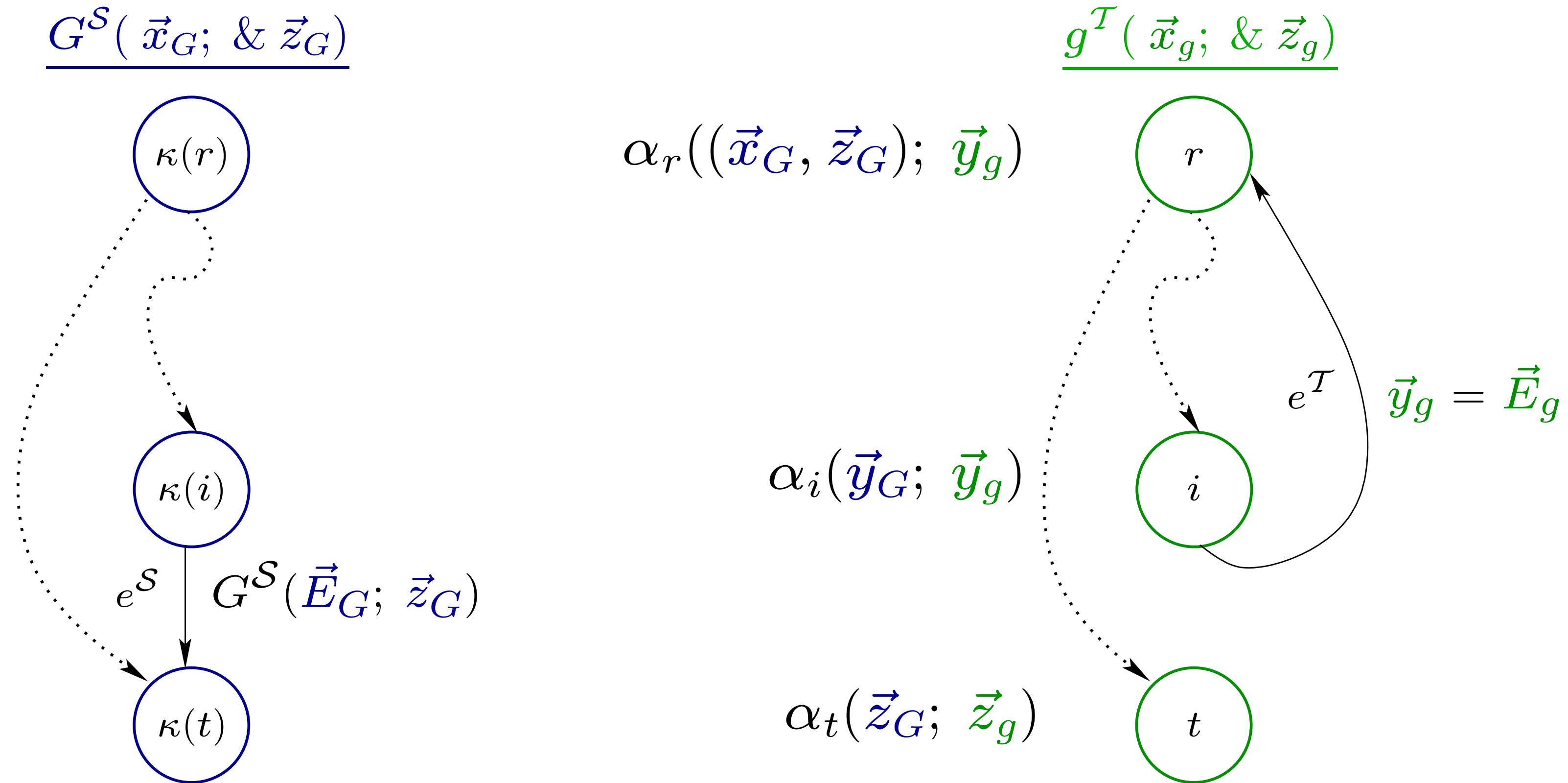
# Translation Verification Conditions: Inlining and TRE

Inlining and Tail-Recursion Elimination(TRE) introduce situations in which the source code contains a call edge that corresponds to a subgraph in the target.

In this case, we prove the translation by “stepping into” the procedure call on the source.

# TRE Verification Conditions

*Definition* : A call edge $(i, t)$ of procedure $G(\vec{x};\ \&\ \vec{z})$ is a TRE candidate if it is labeled by $G(\vec{E};\vec{z})$ and $t$ is the exit node of the procedure.




$$\mathcal{VC}_{call:}\ \alpha_i(\vec{y}_G;\ \vec{y}_g) \ \rightarrow\ \alpha_r((\vec{E}_G, \vec{z}_G);\ \vec{E}_g)$$

# TRE Verification Conditions: Justification

- Step in: When $\mathcal{S}$ steps deeper into the recursion level and $\mathcal{T}$ steps deeper into the iteration level:
  - The validity of the data abstraction is guaranteed by the TRE Verification Condition.
- Step out: After both programs have reached the full recursion (iteration) depth, and $\mathcal{S}$ executes the return transitions:
  - The data abstraction is preserved by the returns since the formal parameters passed by reference: $\vec{z}$, are used as the actual parameters in the tail call.
  - The observations of $\mathcal{S}$ and $\mathcal{T}$ are stuttering equivalent since only a finite number of return transitions is executed.

# Factorial Example: TRE Verification Condition

$\mathcal{S}$

$\underline{MAIN()}$: 1 $\xrightarrow{read(A)}$ 2 $\xrightarrow{(C,B) := (5,1)}$ 3 $\xrightarrow{MFAC(3*A, B)}$ 4 $\xrightarrow{write(B+C)}$ 5

$\underline{MFAC(X; \&Z)}$: 6 $\xrightarrow{(X > 1) \rightarrow Z := Z * X}$ 7 $\xrightarrow{MFAC(X-1, Z)}$ 8; 6 $\xrightarrow{(X \leq 1)?}$ 8

$\alpha_7 : (X = x) \wedge (Z = z)$
$\alpha_6 : (X = x) \wedge (Z = z)$

$\alpha_7 \quad \rightarrow \alpha_6 [X \mapsto X - 1;\ x \mapsto x - 1]$

$\Longleftrightarrow$

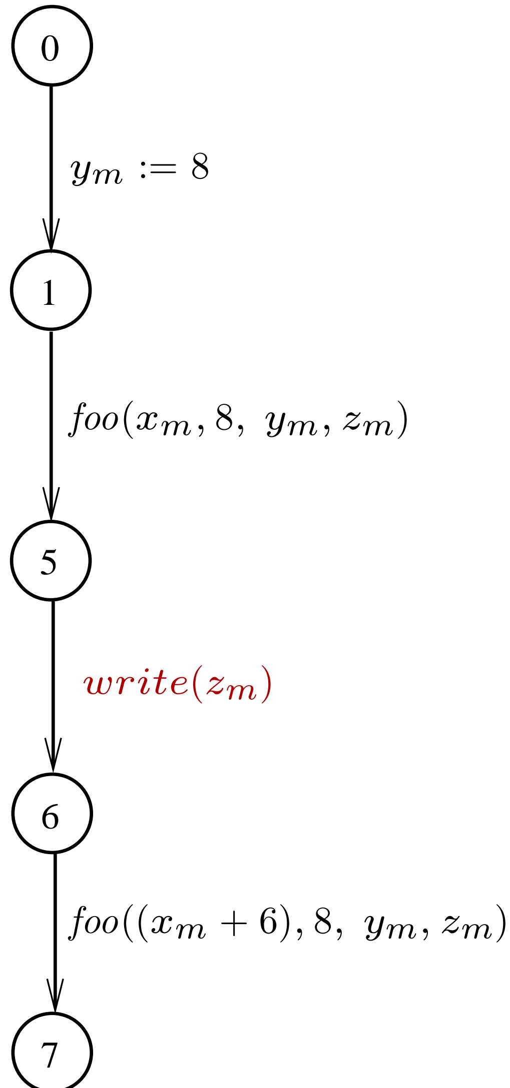
$(X = x) \wedge (Z = z) \rightarrow (X - 1 = x - 1) \wedge (Z = z)$

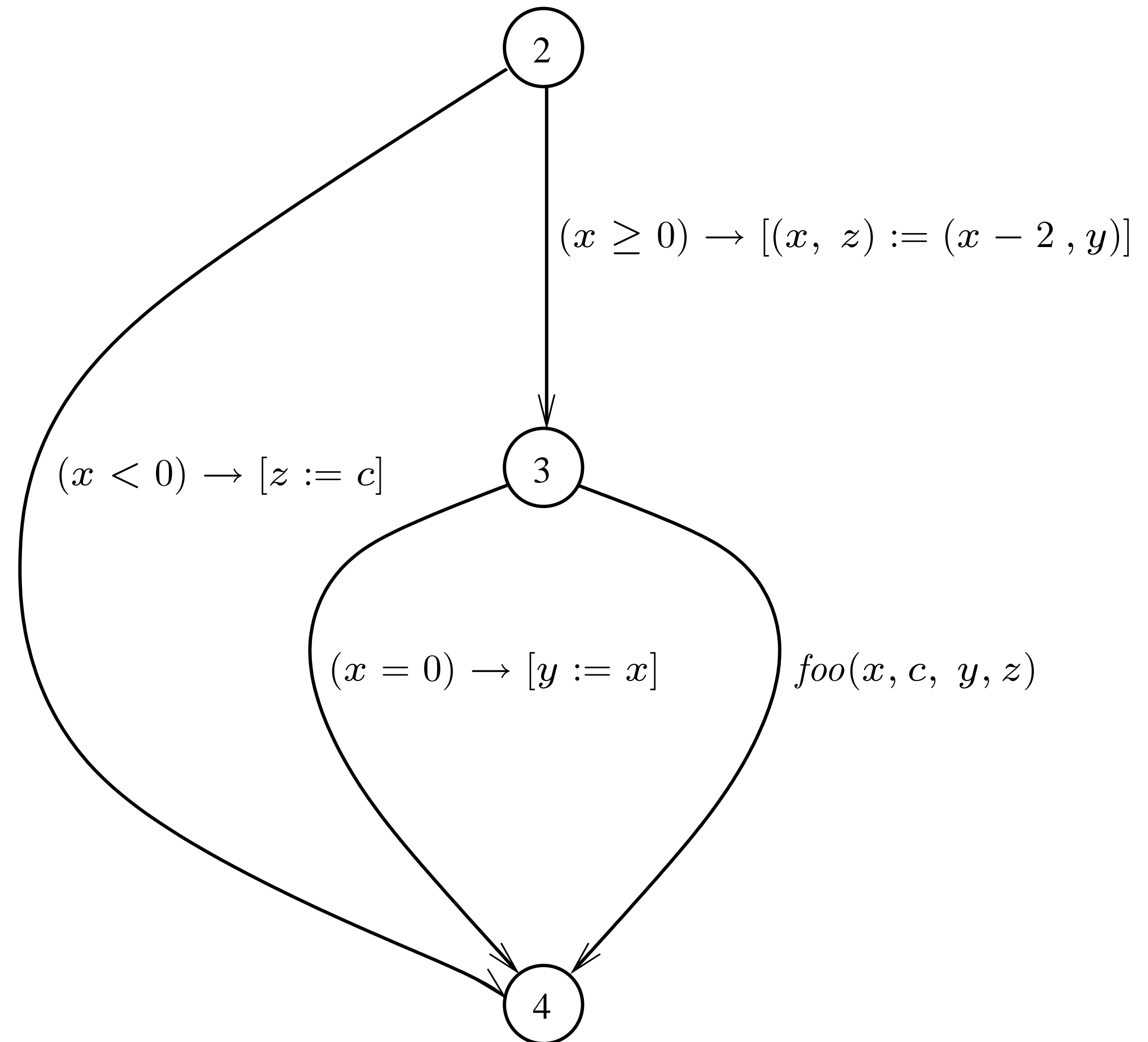
$\underline{mfac(x; \&z)}$: 6 $\xrightarrow{(x > 1) \rightarrow z := z * x}$ 7; 7 $\xrightarrow{x := x - 1}$ 6; 6 $\xrightarrow{(x \leq 1)?}$ 8

$\mathcal{T}$

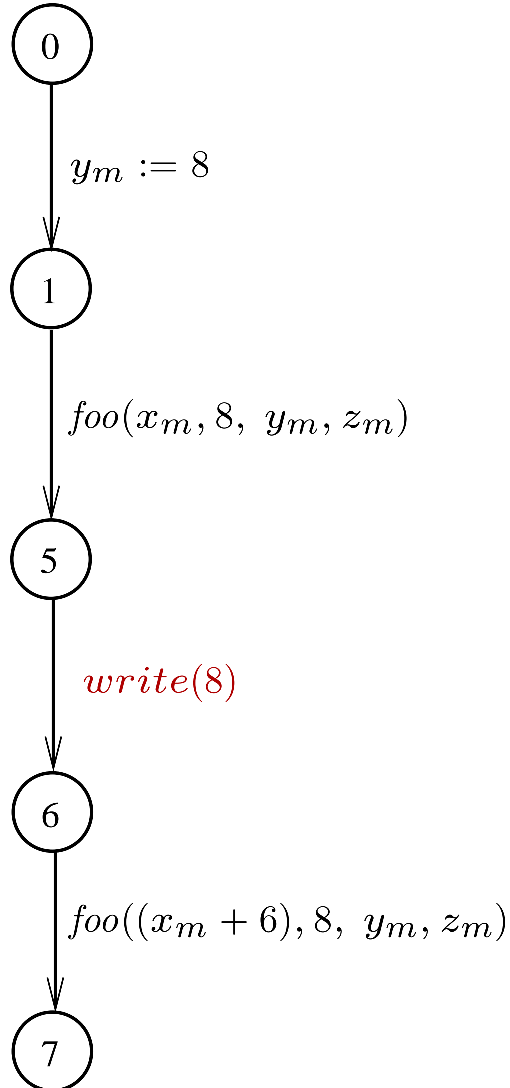
$\underline{main()}$: 1 $\xrightarrow{read(a)}$ 2 $\xrightarrow{(k,b) := (3*a, 1)}$ 3 $\xrightarrow{mfac(k,b)}$ 4 $\xrightarrow{write(b+5)}$ 5

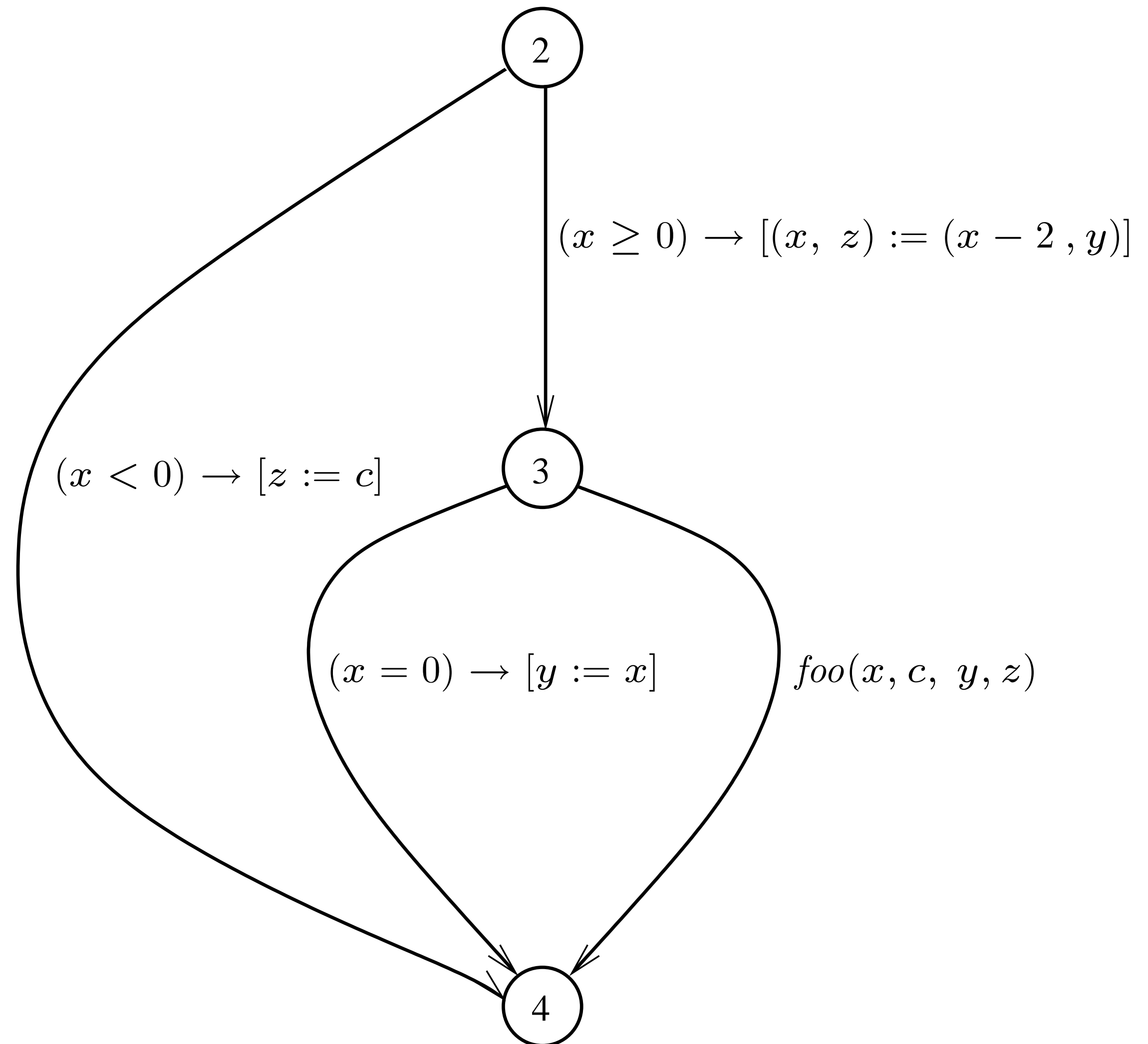
# Example of Constant Propagation: Source



# Example of Constant Propagation: Target



# Inductive Assertion Network for Global Constant Propagation

In order to prove translation in presence of interprocedural constant propagation, the verification conditions have to be strengthened with auxiliary assertion network that:

- is strong enough; for example, it allows to prove the translation verification condition associated with the write transitions from $l_5$ to $l_6$:

$$\varphi_5 \wedge \alpha_5 \rightarrow \alpha_6 \wedge (z_m = 8)$$

- is inductive; thus, it is self-sufficient for proving that all its assertions are invariants.

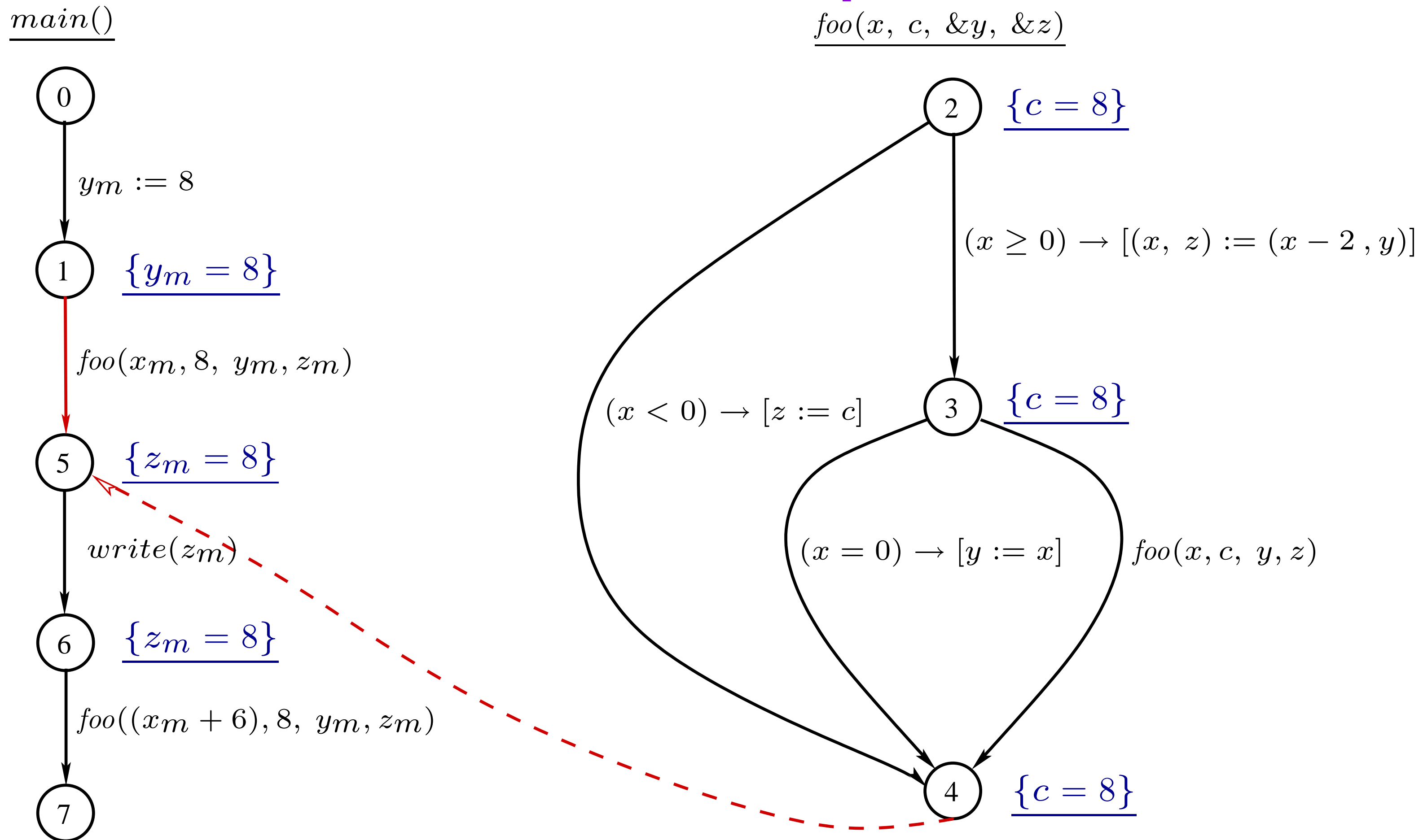
# First Attempt

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

The resulting assertion network:

- is strong enough to prove translation but
- may not be inductive.

# Back to our Example




For example, the return verification condition for call edge $(1, 5)$ does not hold:

$$
\begin{array}{llllllll}
\mathcal{VC}_{ret}: & \varphi_1 & \wedge & \varphi_4 & \rightarrow & \varphi_5[z_m \mapsto z] & \Leftrightarrow \\
 & y_m = 8 & \wedge & c = 8 & \rightarrow & z = 8 &
\end{array}
$$

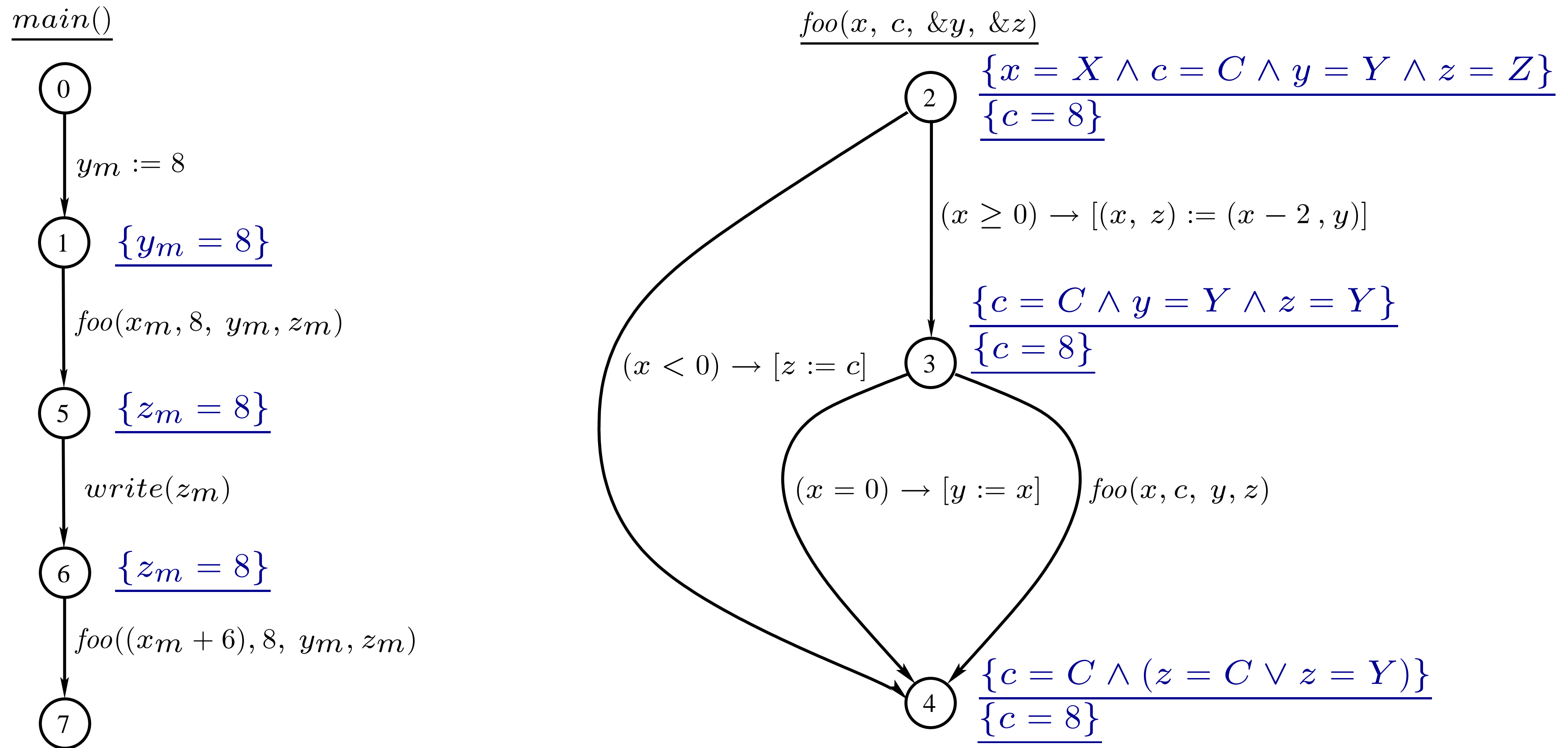
# Solution

- Fortunately, the algorithm presented in Precise Interprocedural Dataflow Analysis with Applications to Constant Propagation by M. Sagiv, T. Reps, and S. Horwitz not only provides a solution to the constant propagation problem, but also finds a set of environment transformers. We are using this additional information to strengthen our network so that it would be inductive.

- The environment transformers are represented as a set of functions, where each 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.

- They relate the values of variables at the beginning of each procedure to the values of the variables at each procedure location.

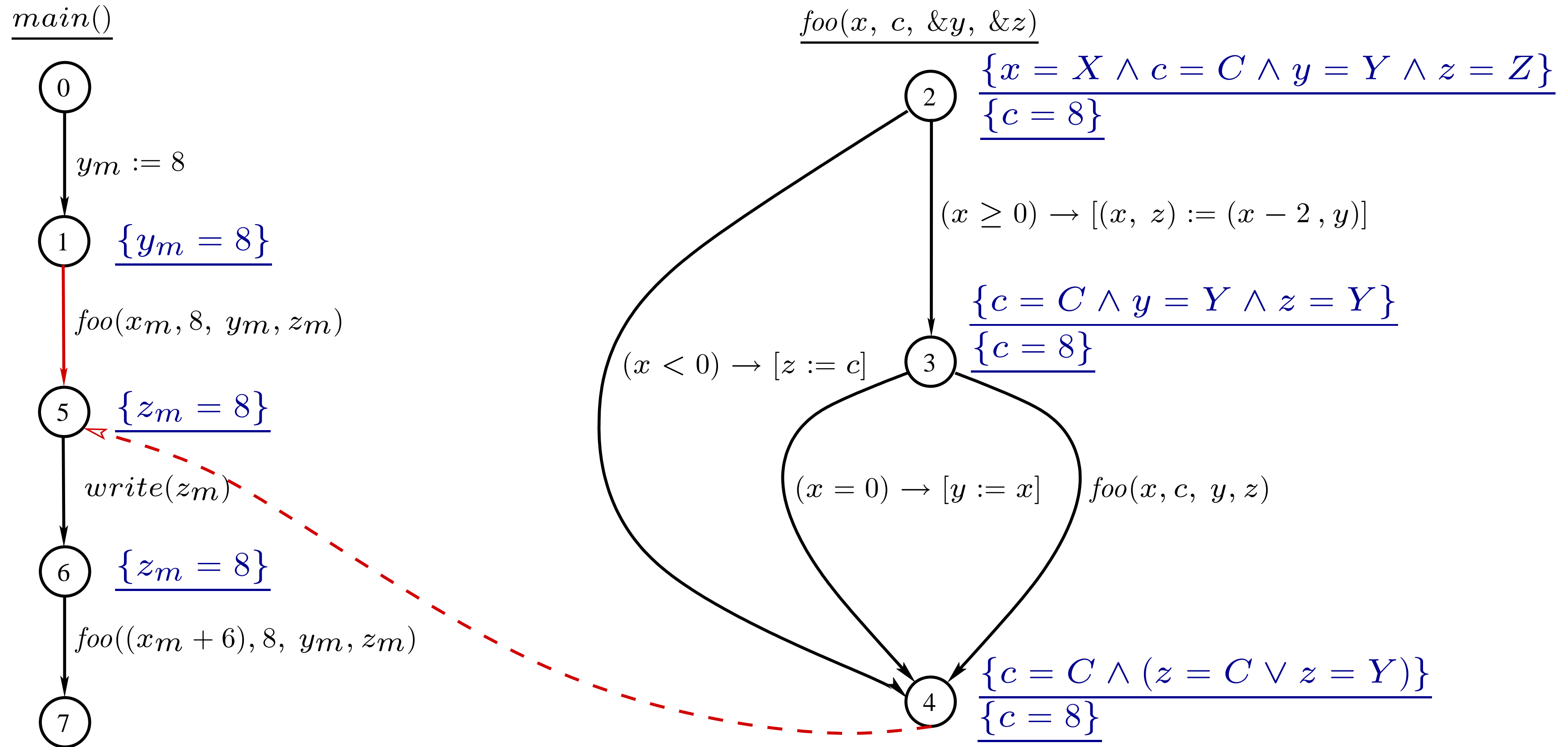
# Resulting Inductive Assertion Network




The list of environment transformers computed for procedure *foo*:

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

# Resulting Inductive Assertion Network




Let's show that the return verification condition for call edge $(1, 5)$ holds:

$$
\begin{array}{llllll}
\mathcal{VC}_{ret}: & \varphi_1 & \wedge & \varphi_4[(C, Y) \mapsto (8, y_m)] & \to & \varphi_5[z_m \mapsto z] & \Leftrightarrow \\
& y_m = 8 & \wedge & c = 8 \wedge (z = 8 \vee z = y_m) \wedge c = 8 & \to & z = 8 &
\end{array}
$$

# Conclusion and Future Work

*Completeness :*

- Since the approach deals with infinite state systems, we cannot hope to have a complete method for proving translation without imposing any restrictions on the transformer.
- However, because the focus is only on compiler optimizations, the number of false alarms can be drastically minimized or even eliminated.

*Implementation :*

We are currently developing a tool that verifies the optimizations performed by LLVM compiler and uses CVC3 as the back-end theorem prover.

*Extensions to the framework :*

- aliasing (in the interprocedural context),
- dynamic memory allocation,
- exceptions.