Symbolic Execution Based Data Flow Analysis for Optimizing Compilers: Proof of Semantic Equivalence of a Program and Code Generated from the Symbolic Execution Based Data Flow Analysis

Emre Kultursay, Kemal Ebcioğlu and Mahmut Kandemir

Department of Computer Science and Engineering
Technical Report
CSE-09-009

August 4, 2009
Abstract
In this technical report, the proof of equivalence of a program with the code generated from it using symbolic execution based data flow analysis is presented.

1. Definitions

Control Flow Graph (CFG): A directed graph whose nodes are primitive statements such as arithmetic, load, store, two-way branch, and two-way join. Its edges represent transitions from a statement to another statement as a result of executing the first statement.

State: A state $\sigma$ (as defined in operational semantics) is a mapping from (address) numbers to (value) numbers that represents the memory.

Extended state: An extended state $\Sigma$ is a pair $(e, \sigma)$, where $e$ is an edge on the control flow graph and $\sigma$ is a program state.

Logical Assertion: A logical assertion (LA) is a pair $(P, M)$, where $P$ is a symbolic predicate expression and $M$ is a statement, both expressed in extended first order logic. After symbolic execution, a logical assertion is associated with every edge in the control flow graph. The statement $M$ is treated as a mapping from address expressions to value expressions written in terms of symbolic variables.

Code Generation: Define $f$ to be a function that takes a logical assertion $L$ and a program state $\sigma$. From the logical assertion, it generates a program $P = f(L, \sigma)$. The iteration counts of loops that are not terminated yet do not exist in $L$. The state $\sigma$ is only used to look up the values of these free loop index variables, providing iteration counts for loops that are not terminated yet.

CFG Execution: Define $executeG$ to be a function that takes an input extended state $\Sigma$, a CFG $G$, and an integer $k$ ($k \geq 0$) to execute $G$ for $k$ instructions starting from initial state $\Sigma$. It generates an extended output state $\Sigma' = executeG(\Sigma, P, k)$.

Program Execution: Define $executeP$ to be a function that takes a state $\sigma$ and a program $P$ to execute $P$ until completion starting from $\sigma$. It generates an output state $\sigma' = executeP(\sigma, P)$.

2. Claim
Let $P$ be a program and $G$ be the control flow graph obtained from $P$. Let $e_0$ be the entry edge of $G$ and let $\sigma_0$ be any state. Assume that starting from the extended state $(e_0, \sigma_0)$ and executing $k$ statements over $G$ results in an extended state $(e_1, \sigma_1)$, where $0 \leq k$. Let $L$ be the logical assertion on edge $e_1$ after performing symbolic execution based data flow analysis on the CFG $G$. Let $P'$ denote the program obtained from $L$. Then, (1) starting from $\sigma_0$ and executing $P'$ results in state $\sigma'$, where $\sigma' = \sigma_1$; and (2) evaluating the predicate of $e_1$ at state $\sigma_1$ results in true.

Note: Let $G'$ be the CFG obtained from $P'$. The edge of $G'$ reached as a result of executing $P'$ is irrelevant since the control flow graph $G'$ is optimized.

3. Assumptions
- We are not considering dead variables that are eliminated as a result of symbolic execution. Actually, the states reached are not equal but are equivalent in terms of only live variables.
- We are considering only single entry, single exit loops. This proof can easily be generalized to multiple entry multiple exit loops.
• We treat registers as special, reserved memory locations that can only be accessed through the name of the register. Therefore, a register can never be aliased with another memory location. For instance, for a register $r_1$, $M[r_1]$ describes the contents of register $r_1$.

• We assume that expression simplification is a valid transformation. As a result, execution of a program obtained from a logical assertion reaches the same state as the program obtained from its simplified version:

$$executeP(\sigma_0, f(simplify(P, E)), \sigma) = executeP(\sigma_0, f(E, \sigma)),$$

for any $\sigma, \sigma_0$.

4. Notation
Let $val$ and $addr$ be numbers representing a value and an address, respectively. Let $vexpr$, $aexpr$, $pred$, $expr$, $expr_1$, and $expr_2$ be symbolic expressions.

• $\sigma(addr)$: This returns the value of address $addr$ in state $\sigma$.
• $\sigma[val/addr]$: In state $\sigma$, the contents of address $addr$ is updated with value $val$.
• $L(aexpr)$: In logical assertion $L$, look up the value expression for the address expression $aexpr$.
• $L[vexpr/aexpr]$: In logical assertion $L$, update the contents of address expression $aexpr$ with value expression $vexpr$.
• $simplify(pred, expr)$: Simplify the symbolic expression $expr$ assuming that $pred$ holds.
• $simplify(bexpr)$: Simplify the symbolic Boolean expression $bexpr$.
• $expr_1 \oplus expr_2$: Symbolic addition of two symbolic expressions.
• When the predicate is not relevant, logical assertions and expression maps are used interchangeably. As a result, $L(aexpr)$ and $L[vexpr/aexpr]$ are equivalent.
• When there are no free index variables of interest in a logical assertion $L$, we omit the second parameter $\sigma$ to the code generation function $f$ and write $f(L)$.
• $eval(P, \sigma)$: Evaluate the predicate $P$ using the values for the symbolic variables from the state $\sigma$ and return a Boolean value. Note that $eval$ is distributive over logical $\land$ and $\lor$ operators.

5. Extended First Order Logic and Operations on Logical Assertions

5.1 Extended First Order Logic
In order to simplify the complexity of representation of sequential execution, we extend the first order logic with two operators: (1) a *semicolon* ($;$) operator that combines two statements; and (2) a *for* operator that combines multiple statements with an index variable over a range given as a parameter (e.g., $for(0 \leq j \leq N) \{L(j)\}$). A logical assertion can be parsed with the following grammar:

$$LA ::= \begin{cases} subLA(;LA) \mid \epsilon \\ if(bexpr) then LA else LA endif (;LA) \end{cases}$$

$$subLA ::= unit \land unit^*$$

$$unit ::= M[aexpr] == expr \mid (for | forall)(0 \leq I \leq I_{FINISH}() - 1) \{LA\}$$

5.2 Operations on Logical Assertions
In this section, we describe the look-up and update operations on logical assertions. Let $A$, $A'$ be symbolic address expressions and $v$, $v_1, v_2$ be symbolic value expressions.

1. Although a LA is an extended first order statement, it can be considered as a mapping from address expressions to value expressions.

2. A LA is composed of a number of address expression to value expression mappings in the form: $M[aexpr] == vexpr$. These mappings are bonded by two types of connectors: (1) a commutative $\land$ ($\land$) operator, (2) a non-commutative *semicolon* ($;$) operator. A LA can be treated as a set of sub-LAs connected by *semicolons*. Logical assertions can also contain *if* statements that multiplex two logical assertions into one.

3. When a look up for a symbolic address expression $A$ on a logical assertion $L$, written as $L(A)$, is requested, $L$ is assumed to be true and it is searched for a sub-assertion of the form $\{M[A] == v\}$ (with an exact match of the left hand side). The sub-LAs connected with *semicolons* are searched from right to left (Note that the *for* operator is a generalization of the *semicolon* operator). This look up order ensures that when there are multiple sub-assertions that provide a value for $A$, the value returned will be its most recent update. If $A$ does not exist in the logical assertion, then the following expression is returned for $M[A]$: for all $A'$ such that we cannot prove $A \neq A'$,

$$if(A = A') then L(A') else M[A]$$
Example: Consider the following logical assertion $L$:

$$L = (true, M[a] == 0; M[b] == 1)$$

Then, if we look up the value for address expression $c$, we get:

$$L(c) = if (b == c) then 1 else if (a == c) then 0 else M[c].$$

4. Alias analysis guarantees the following: For a sub-assertion $L = \{M[A_1] == v_1 \wedge M[A_2] == v_2\}$, and for some $\sigma$,

$$if (L_o(A_1) = L_o(A_2)) then \sigma(v_1) = \sigma(v_2),$$

where $L_o$ denotes the replacement of all symbolic variables in $L$ with their values in $\sigma$. This means that in the some state $\sigma$, if two address expressions evaluate to the same concrete address, then their value expressions must evaluate to equal concrete values.

5. When an update $L[v/A]$ on logical assertion is requested, firstly a new assertion of the form $\{L; M[A] == v\}$ is created. Then, we check the validity of converting the $semicolon$ operator into an $and$ operator with a dependence analysis. If it is possible, then the any existing $M[A] == v_0$ mapping in the last sub-assertion of $L$ is removed, and the value expressions for all addresses that might be aliased with $A$ are updated with conditional expressions.

Example: Consider the following logical assertion over the symbolic variables $a$, $b$, and $x$:

$$L = \{\{M[a] == 2 * x \wedge M[b] == 2 * x + 1\}; \{M[a] == 3 * x\}\}$$

In this example, for illustrative purposes, assume (1) the $semicolon$ operator connecting the initial two sub-LAs cannot be converted into an $and$ operator; and (2) the variables $a$ and $b$ are not aliased. Then,

$$L(a) = 3 * x$$
$$L(b) = 2 * x + 1$$
$$L[5/a] = \{M[a] == 2 * x \wedge M[b] == 2 * x + 1; M[a] == 5\}$$
$$L[7/b] = \{M[a] == 2 * x \wedge M[b] == 2 * x + 1; M[a] == 3 * x; M[b] == 7\}$$
$$= \{\{M[a] == 2 * x \wedge M[b] == 2 * x + 1; M[a] == 3 * x \wedge M[b] == 7\}\}$$

6. Similarly, in special occasions indicated by dependence analysis, a sequential $for$ quantifier in a logical assertion can be converted into a standard universal quantifier ($forall$).

6. Property

Let $L_1$ and $L_2$ be two logical assertions and $\sigma_0$ be a state. Then,

$$executeP(\sigma_0, f(L_1; L_2)) = executeP(\sigma_0, f(L_1); f(L_2))$$
$$= executeP(executeP(\sigma_0, f(L_1)), f(L_2)).$$

This property states that if a logical assertion $L$ can be separated into two sub-assertions $L_1$ and $L_2$ connected with a $semicolon$, then executing program obtained from $L$ is semantically equivalent to executing program obtained from $L_1$ and later continuing by executing the program obtained from $L_2$. Note however that,

$$f(L_1; L_2) \neq f(L_1); f(L_2),$$

since the program on the left hand side is an optimized version of the program on the right hand side. These programs are only semantically equivalent, shown as:

$$f(L_1; L_2) \equiv f(L_1); f(L_2).$$

A special case of this property can be written for the update operation:

$$executeP(\sigma_0, f(L[v/a])) = executeP(executeP(f(L)), f(M[a] == v)).$$

This property is used extensively throughout the inductive step of the proof.

7. Modifications on the Control Flow Graph Before Processing

Before symbolic execution, the loop hierarchy of the control flow graph is identified and some modifications on the CFG nodes corresponding to loops are performed. Let $L$ represent a loop in the program, then the following modifications are performed on $L$:

1. A unique index variable $I$ is defined, and is assigned to $L$. This variable is inactive before the loop is entered.
2. A new CFG node called “loop entry node” on the forward entry edge of $L$ is created. It contains the statement $I = 0$.
3. A new CFG node called “loop back edge node” on the loop back edge of $L$ is created. It contains the statement $I = I + 1$.
4. A new CFG node called “loop exit node” on the loop exit edge of $L$ is created. It contains the statement $I = I - 1$.
5. A function $f_{finish}()$ that represents the number of iterations this loop is executed is defined. Typically, this function is statically uncomputable. It is a function of the outer loop index variables and other variables that are used inside the loop that affect the number of iterations.
These new CFG nodes are simple, single entry single exit nodes. Further, two already existing nodes of the loop are given special names: (1) Loop Exit Branch Node is the only node inside the loop with one of its outgoing edges exiting the loop; and (2) LoopEntryJoinNode is the only node that has an incoming edge from a node outside the loop. Note that any single entry, single exit loop can be transformed into this structure. A loop can be treated as a set of nodes combined to form a virtual super-node called a loop box, as shown in Figure 1.

![Figure 1](image1.png)

**Figure 1.** Nodes introduced or identified for each loop in a program control flow graph. The whole loop can be considered as a single entry, single exit virtual box.

### 8. Description of Program Semantics

In order to show the equivalence of CFG execution and generated program execution, we first describe the following:

1. Operational semantics of execution over the CFG,
2. Semantics of extracting logical assertions using symbolic execution over the CFG,
3. Code generation from logical assertions.

The two semantics will be explained for the following 10 different types of nodes in a CFG: (1) arithmetic; (2) load; (3) store; (4) conditional jump; (5) 2-way join; (6) loop entry join node; (7) loop exit node; (8) loop entry node; (9) loop exit branch node; and (10) loop back edge node. These node types are shown in Figure 2. Code generation will be explained using a context free grammar. Throughout the rest of this report, the incoming and outgoing edges of a node under interest are labeled as $e_{in}$ and $e_{out}$, respectively. When there are two incoming or outgoing edges, they are identified with integers 1 and 2, such as $e_{in1}$, $e_{in2}$, $e_{out1}$, $e_{out2}$.

#### 8.1 Operational Semantics

##### 8.1.1 Arithmetic Instruction

Input Extended State:

\[ (e_{in}, \sigma_{in}) \]

Output Extended State:

\[ (e_{out}, \sigma_{in}[\sigma_{in}(r2) + \sigma_{in}(r3)/r1]) \]
8.1.2 Load Instruction
Input Extended State:
\[(e_{in}, \sigma_{in})\]
Output Extended State:
\[(e_{out}, \sigma_{in}[\sigma_{in}(r2)/r1])\]

8.1.3 Store Instruction
Input Extended State:
\[(e_{in}, \sigma_{in})\]
Output Extended State:
\[(e_{out}, \sigma_{in}[\sigma_{in}(r2)/\sigma_{in}(r1)])\]

8.1.4 Conditional Jump
Input Extended State:
\[(e_{in}, \sigma_{in})\]
Output Extended State:
\[\text{if} (\sigma_{in}(\text{cond})) \text{ then } (e_{out1}, \sigma_{in}) \text{ else } (e_{out2}, \sigma_{in})\]

8.1.5 Two-way Join
Execution starts at state \(\sigma_{in}\), and is entering a two way join node. This node acts as a no operation. Input edge \(e_{in}\) can be \(e_{in1}\) or \(e_{in2}\), the output edge is the same for both.

Input Extended State:
\[(e_{in1}, \sigma_{in}) \text{ or } (e_{in2}, \sigma_{in})\]
Output Extended State:
\[(e_{out}, \sigma_{in})\]

8.1.6 Loop Entry Join Node
From operational semantics point of view, a loop entry node is a no operation, similar to the two-way join node.
Input Extended State:
\[(e_{in}, \sigma_{in})\]
Output Extended State:
\[(e_{out}, \sigma_{in})\]

8.1.7 Loop Exit Node
Input Extended State:
\[(e_{in}, \sigma_{in})\]
Output Extended State:
\[(e_{out}, \sigma_{in}[−1/1])\]
8.1.8 Loop Entry Node
Input Extended State: 
\((e_{in}, \sigma_{in})\)
Output Extended State: 
\((e_{out}, \sigma_{in}[0/I])\)

8.1.9 Loop Exit Branch Node
Input Extended State: 
\((e_{in}, \sigma_{in})\)
Output Extended State: 
\((e_{in1}, \sigma_{in}) \text{ or } (e_{in2}, \sigma_{in})\)
where,
\(e_{in1}\): Edge exiting the loop,
\(e_{in2}\): Edge that stays within the loop.

8.1.10 Loop Back Edge Node
Input Extended State: 
\((e_{in}, \sigma_{in})\)
Output Extended State: 
\((e_{out}, \sigma_{in}[(\sigma_{in}(I) + 1)/I])\)

8.2 Symbolic Execution Semantics
For an input logical assertion \(L_{in} = (P_{in}, M_{in})\), \(P_{in}\) stands for the predicate and \(M_{in}\) is equivalent to a mapping from address expressions to value expressions. Note that registers are treated as special memory locations which have reserved names. Let \(reg\) be a register name (e.g., \(r1\)), \(addr\) be a symbolic expression that represents an address (e.g., \(2 \times x + 1\)), and \(val\) be a symbolic expression that represents a value (e.g., \(3 \times y + 2\)). We will use the following semantics for look up and update operations on logical assertions:

- \(M_{in}(reg)\): In \(M_{in}\), get contents of register \(reg\).
- \(M_{in}(addr)\): In \(M_{in}\), get contents of the memory address \(addr\).
- \(M_{in}(val/reg)\): In \(M_{in}\), update the register \(reg\) contents with value expression \(val\).
- \(M_{in}(val/addr)\): In \(M_{in}\), update the memory address expression \(addr\) contents with value expression \(val\).

8.2.1 Arithmetic Instruction
Input LA: 
\((P_{in}, M_{in})\)
Output LA: 
\((P_{in}, M_{in}[simplify(P_{in}, M_{in}(r2) \oplus M_{in}(r3))/r1])\)

8.2.2 Load Instruction
Input LA: 
\((P_{in}, M_{in})\)
Output LA: 
\((P_{in}, M_{in}[M_{in}(M_{in}(r2))/r1])\)

8.2.3 Store Instruction
Look up the contents of register \(r1\) in \(M_{in}\) to obtain an address expression \(aexpr\) and contents of register \(r2\) to obtain a value expression \(vexpr\). In \(M_{in}\), for every address expression \(aexpr' \neq aexpr\), update its contents with: 
\(simplify(P_{in}, if(aexpr' == aexpr) then vexpr else M_{in}(aexpr'))\) to obtain \(M'_{in}\). Then, update contents of \(aexpr\) with \(vexpr\) in \(M'_{in}\).
Input LA: 
\((P_{in}, M_{in})\)
Output LA: 
\((P_{in}, M'_{in}[M_{in}(r2)/M_{in}(r1)])\),
where, \(M'_{in}\) = The expression map obtained from \(M_{in}\) after performing modifications that handle all possible addresses that may be aliased with the address \(M_{in}(r1)\).

8.2.4 Conditional Jump
Input LA: 
\((P_{in}, M_{in})\)
Output LAs: 
\((simplify(P_{in} \land M_{in}(cond)), M_{in})\)
\((simplify(P_{in} \land \neg M_{in}(cond)), M_{in})\)
8.2.5 Two-way Join
Input LAs:
\[(P_{i1}, M_{i1}), (P_{i2}, M_{i2})\]
Output LA:
\[(P_{out}, M_{out})\]
where,
\[P_{out} = \text{simplify}(P_{i1} \lor P_{i2})\]
\[M_{out} = \text{simplify}(P_{out}, (P_{i1} \land M_{i1}) \lor (P_{i2} \land M_{i2}))\]
Note that since \(P_{i1} \land P_{i2} = \text{false}\), we can write:
\[M_{out}(x) = \text{simplify}(P_{out}, \text{if}(P_{i1}) \text{then} M_{i1}(x) \text{else} M_{i2}(x)),\]
or equivalently,
\[M_{out}(x) = \text{simplify}(P_{out}, \text{if}(P_{i2}) \text{then} M_{i2}(x) \text{else} M_{i1}(x)),\]
for any symbolic expression \(x\).

8.2.6 Loop Entry Join Node
Let \(LB(j)\) denote the logical assertion obtained from symbolic execution of the loop body (after performing induction variable substitution). Define the function \(\text{delete}(M, M[I] == 0)\) as the expression map equal to \(M\) except that the assertion \(M[I] == 0\) is deleted.
Input LAs:
\[(P_{i1}, M_{i1}), (P_{i2}, M_{i2})\]
Output LA:
\[(P_{out}, M_{out}),\]
where,
\[P_{out} = P_{i1} \land (0 \leq I \leq FINISH() - 1),\]
\[M_{out} = \{M'_{i1}; \text{for}(0 \leq j \leq I - 1) \{LB(j)\}\},\]
\[M'_{i1} = \text{delete}(M_{i1}, M[I] == 0),\]
A loop entry join node is treated as a special two-way join node. The loop body is represented with an assertion \(LB\). In \(LB\), all induction variables have been substituted with equivalent expressions in terms of the imaginary loop index variable \(I\) of this loop. In order to obtain \(M_{out}\), first, using the \(\text{delete()}\) function, \(I\) is removed from \(M_{i1}\) to obtain \(M'_{i1}\), so the exact value of \(I\) is unknown. The logical assertion contains only its limits: \(0 \leq I \leq FINISH() - 1\). Then, an extra assertion with a sequential for quantifier is added to \(M_{out}\) to represent first \(I\) iterations of this loop. Note that we are not propagating any information from the loop back edge, but the back edge contains an assertion that represents the first \(I\) iterations of the loop:
\[M_{i2} = \{M'_{i1}; \text{for}(0 \leq j \leq I - 1)\{LB(j)\}\}.\]

8.2.7 Loop Exit Node
We define the function \(\text{replace}(M, v_{in}, v_{out})\) such that it replaces all instances of \(v_{out}\) with \(v_{in}\) in \(M\).
Input LA:
\[(P_{in}, M_{in})\]
Output LA:
\[(P_{out}, M_{out}),\]
where,
\[P_{out} = \text{delete}(P_{in}, 0 \leq I \leq FINISH() - 1),\]
\[M_{out} = \{\text{replace}(M_{in}, FINISH(), I) \land M[I] == -1\},\]
with
\[FINISH() = \min\{j | j \geq 0 \land \text{cond}(j) = \text{false}\}.\]
An overloaded version of the \(\text{delete()}\) function is used to reduce the conditions inside the predicate. The subexpression to be deleted is replaced with the value \text{true}. The predicate no longer contains a range for \(I\). In the expression map, all instances of \(I\) are replaced with the function \(FINISH()\). The value of \(I\) at loop exit is set to \(-1\).

8.2.8 Loop Entry Node
Input LA:
\[(P_{in}, M_{in})\]
Output LA:
\[(P_{in}, M_{in}[0/I]),\]
8.2.9 Loop Exit Branch Node

Input LA:

\[(P_{in}, M_{in})\]

Output LAs:

\[(P_{out1}, M_{out1}), (P_{out2}, M_{out2})\],

where,

\[P_{out1} = P_{in}\]
\[P_{out2} = P_{in} \land \neg \text{cond}(I)\]
\[M_{out} = M_{in}.\]

Infinite execution of the loop can be treated as yet another exit of the loop. However, since non-termination is a don’t care condition, we can use its predicate to simplify the exit condition of the loop. Therefore, we are not adding the loop exit condition to the predicate on the loop exit edge. When there are multiple exits of a loop, the condition on this don’t care edge can be used to extend the condition of only one of these exits.

8.2.10 Loop Back Edge Node

Input LA:

\[(P_{in}, M_{in})\]

Output LA:

\[(P_{out}, M_{out})\]

where,

\[P_{out} = P_{in}\]
\[M_{out} = M_{in}[\text{simplify}(M_{in}(I) - 1)/I].\]

The loop back edge node executes the opposite of the instruction in the node, namely \(I = I - 1\). This replaces all \(I\)s with \((I - 1)\)s.

8.3 Code Generation from Logical Assertions

In this section, we repeat the grammar to parse a logical assertion and demonstrate how we generate code from each rule.

\[
\begin{align*}
L A & ::= \text{subLA}(; L A)? \\
& | \text{if}(bexpr) \text{then LA else LA endif} (; L A)? \\
& | \epsilon \\
\text{subLA} & ::= \text{unit} (\land \text{unit})^* \\
\text{unit} & ::= M[expr] == expr \\
& | (\text{for} | \text{forall}) (0 \leq I \leq I_{FINISH}() - 1) [ L A ]
\end{align*}
\]

1. \text{subLA}(; L A)?
   - \(C_1 = \text{code(\text{subLA})}\).
   - \(C_2 = \text{code(\text{LA})}\).
   - If \text{LA} exists, then output the code: \(C_1C_2\), else output the code: \(C_1\).
   
   Note that since the \text{semicolon (;)} operator is non-commutative, \(C_1\) and \(C_2\) cannot be reordered.

2. \text{if}(bexpr) \text{then LA else LA endif} (; L A)?
   - If \text{LA} exists, then output the program fragment:
     
     \[
     \begin{align*}
     t &= \text{code(bexpr)} \\
     \text{if}(t) & \text{then code(LA)}_1 \text{endif} \\
     \text{else code(LA)}_2 \text{endif} \text{code(LA)}_3
     \end{align*}
     \]
     
     - If \text{LA} does not exist, then the \text{code(LA)} statement should be removed.

3. \epsilon
   - Do not generate any code.

4. \text{unit1} \land \text{unit2}
   - \(C_1 = \text{unit1}\).
   - \(C_2 = \text{unit2}\).
   - Output either:
\[ C_1; \ C_2; \]

or,
\[ C_2; \ C_1; \]

Note that since the and (\(\land\)) operator is commutative, \(C_1\) and \(C_2\) can be reordered.

5. \( M[\text{LHS}] == \text{RHS} \)

- Let \(t, t_2\) be two registers.
- if LHS is a register
  \[ \text{LHS} = \text{code}(\text{RHS}) \]
- else, (LHS is a memory expression)
  \[ t = \text{code}(\text{RHS}) \]
  \[ t_2 = \text{code}(\text{LHS}) \]
  \[ M[t_2] = t \]

6. \( \text{(for} \ | \ \forall) \ (0 \leq I \leq I_{\text{FINISH}}() - 1) \ [\text{LA}] \)

- If \(I_{\text{FINISH}}()\) can be solved statically to give a number of iterations \(N\), the exit condition becomes: \((I < N)\), otherwise, we use the original condition from \(I < I_{\text{FINISH}}()\).
- Output the code:
  \[
  \text{LBL:} \\
  \quad \text{code}(\text{LA}) \\
  \quad t = \text{code}(\text{condition}) \\
  \quad \text{if (}t\text{) then goto LBL}
  \]

Note that after a dependence analysis on loops, parallel loop codes can be generated.

9. Proof
The proof is performed by course of values induction on the number of instructions executed over CFG \(P\). Assuming that the claim in Section 2 holds for all execution trace lengths \(k\) with \(0 \leq k \leq N\), prove that if one more instruction is executed \((N + 1)\) statements,
then the claim still holds.

9.1 Induction Base Case
We will show that the claim holds for program execution trace length of 0 statements. If the original program starts in state entry edge \((e_0, \sigma_0)\), then after executing 0 statements, it terminates at \((e_1, \sigma_1)\), where \(e_1 = e_0\) and \(\sigma_1 = \sigma_0\). The logical assertion corresponding to edge \(e_1\) is \((\text{true}, \{\})\), whose predicate trivially evaluates to \text{true}. The empty assertion generates an empty program \(P'\). When \(P'\) is executed starting from \(\sigma_0\), it ends at state \(\sigma' = \sigma_0\). Therefore, the base case holds trivially.

9.2 Inductive Hypothesis
Let \(P\) be the original program code and \(G\) be the CFG obtained from it. Let \(e_0\) be the entry edge to \(G\). Assume that for any \(k\) such that \(0 \leq k \leq N\), starting from extended state \((e_0, \sigma_0)\), executing \(k\) statements over \(G\) results in the final extended state \((e, \sigma)\). Let \(L\) be the logical assertion on the edge \(e\) and let \(P'\) be the program obtained from \(L\). Then, starting from \(\sigma_0\) and executing \(P'\) results in state \(\sigma'\), where \(\sigma' = \sigma\). We also assume that evaluating the predicate on the incoming edge results in \text{true}.

9.3 Inductive Step
We enumerate all possible cases for the structure of the \(N + 1^{\text{st}}\) statement that is added to the execution trace. Assume that the program is entered from an extended state \(\Sigma_0 = (e_0, \sigma_0)\). Let \(\Sigma_{\text{in}} = (e_{\text{in}}, \sigma_{\text{in}})\) and \(\Sigma_{\text{out}} = (e_{\text{out}}, \sigma_{\text{out}})\) be the extended states before and after the node corresponding to the \(N + 1^{\text{st}}\) statement, respectively. When there are two incoming/outgoing edges, these edges are enumerated with integers 1 and 2.

9.3.1 Arithmetic Instruction
Statement to be executed:
\[
\text{add} \ r1 = r2, r3
\]

Input Edge Logical Assertion:
\[
\text{LA}_{\text{in}} = (P_{\text{in}}, M_{\text{in}})
\]

Output Edge Extended State:
\[
\Sigma_{\text{out}} = (e_{\text{out}}, \sigma_{\text{in}}[\sigma_{\text{in}}(r2) + \sigma_{\text{in}}(r3)/r1])
\]

Output Edge Logical Assertion:
\[
\text{LA}_{\text{out}} = (P_{\text{in}}, M_{\text{in}}[\text{simplify}(P_{\text{in}}, M_{\text{in}}(r2) \oplus M_{\text{in}}(r3))/r1])
\]
Since no loop is entered or exited with this instruction, 

Using the property:

\[ r \]

of the procedure or the last semicolon (e.g.,

Note that all predicate expressions are written in terms of initial values of symbolic variables. These are the values before the entry

Executing the generated program gives the state:

\[ \text{Output Edge Logical Assertion:} \]

\[ \text{Statement to be executed:} \]

9.3.2 Load Instruction

Program Generated from Output Edge LA:

\[ f(LA_{out}, \sigma_{out}) = f(M_{in}[\text{simplify}(P_{in}, M_{in}(r2) \oplus M_{in}(r3))/r1], \sigma_{out}) \]

State obtained from execution of generated program:

\[ \sigma'_{out} = \text{executeP}(\sigma_0, f(LA_{out}, \sigma_{out})) = \text{executeP}(\sigma_0, f(M_{in}[\text{simplify}(P_{in}, M_{in}(r2) \oplus M_{in}(r3))/r1], \sigma_{out})) \]

Using the property described in Section 6, we convert this equation into:

\[ \sigma'_{out} = \text{executeP}(\text{executeP}(\sigma_0, f(M_{in}, \sigma_{in})), f(M[r1] \Rightarrow \text{simplify}(P_{in}, M_{in}(r2) \oplus M_{in}(r3))], \sigma_{out}) \). \]

Since no loop is entered or exited with this instruction, \( \sigma_{in} \) and \( \sigma_{out} \) contain the same set of free variables. As a result, we have \( \forall LA: f(LA, \sigma_{in} = f(LA, \sigma_{out}) \). This gives,

\[ \sigma'_{out} = \text{executeP}(\text{executeP}(\sigma_0, f(M_{in}, \sigma_{in})), f(M[r1] \Rightarrow \text{simplify}(P_{in}, M_{in}(r2) \oplus M_{in}(r3))], \sigma_{in}) \). \]

Note that, the first argument of the outer \( \text{executeP} \) is what we have as the inductive hypothesis:

\[ \sigma_{in} = \text{executeP}(\sigma_0, f(M_{in}, \sigma_{in})). \]

Using this inductive hypothesis:

\[ \sigma'_{out} = \text{executeP}(\sigma_{in}, f(M[r1] \Rightarrow \text{simplify}(P_{in}, M_{in}(r2) \oplus M_{in}(r3))/r1], \sigma_{in})). \]

Then, from the definition of code generation for addition operation, this is equivalent to executing \( r1 = r2 + r3 \) at state \( \sigma_{in} \), which gives:

\[ \sigma_{out} = \sigma_{in}[r2 + \sigma_{in}(r3)]/r1 \]

For the second part of the proof, from the inductive hypothesis, we have:

\[ \text{eval}(P_{in}, \sigma_{in}) = \text{true}, \]

and we need to show that

\[ \text{eval}(P_{out}, \sigma_{out}) = \text{true}. \]

Using the equalities we have on \( P_{out} \) and \( \sigma_{out} \):

\[ \text{eval}(P_{out}, \sigma_{out}) = \text{eval}(P_{in}, \sigma_{in}[r2 + \sigma_{in}(r3)/r1]) \]

Note that all predicate expressions are written in terms of initial values of symbolic variables. These are the values before the entry of the procedure or the last semicolon (e.g., \( r10 \)). As a result, an update to \( r1 \) cannot change the initial values of these variables (e.g., \( r1 = r1 + 1 \) gives the assertion \( M[r1] = M[r10] + 1 \)). Therefore, we obtain:

\[ \text{eval}(P_{out}, \sigma_{out}) = \text{eval}(P_{in}, \sigma_{in}) = \text{true}. \]

9.3.2 Load Instruction

Statement to be executed:

\[ r1 = M[r2] \]

Input Edge Logical Assertion:

\[ LA_{in} = (P_{in}, M_{in}) \]

Output Edge Extended State:

\[ \Sigma_{out} = (e_{out}, \sigma_{in}[\sigma_{in}(r2)/r1]) \]

Output Edge Logical Assertion:

\[ LA_{out} = (P_{in}, M_{in}[M_{in}(r2)/r1]) \]

Program Generated from Output Edge logical assertion \( LA_{out} \):

\[ f(LA_{out}, \sigma_{out}) = f(M_{in}[M_{in}(r2)/r1], \sigma_{out}) \]

Executing the generated program gives the state:

\[ \sigma'_{out} = \text{executeP}(\sigma_0, f(LA_{out}, \sigma_{out})) = \text{executeP}(\sigma_0, f(M_{in}[M_{in}(r2)/r1], \sigma_{out})) \]

Using the property:

\[ \sigma'_{out} = \text{executeP}(\text{executeP}(\sigma_0, f(M_{in}, \sigma_{in})), f(M[r1] \Rightarrow M_{in}(r2), \sigma_{out})). \]

Since no loop is entered or exited with this instruction, \( \sigma_{in} \) and \( \sigma_{out} \) contain the same set of free variables. As a result,

\[ \sigma'_{out} = \text{executeP}(\text{executeP}(\sigma_0, f(M_{in}, \sigma_{in})), f(M[r1] \Rightarrow M_{in}(r2), \sigma_{in})). \]
Note that, from the inductive hypothesis:
\[ \sigma_{in} = \text{execute}(\sigma_0, f(M_{in}, \sigma_{in})). \]
Therefore,
\[ \sigma'_{out} = \text{execute}(\sigma_{in}, f(M[r1] == M_{in}(r2), \sigma_{in})). \]
Then, we get:
\[ \sigma'_{out} = \sigma_{in}[\sigma_{in}(r2)/r1] = \sigma_{out}. \]
Proof for the evaluation of the predicate at the outgoing edge is same as the arithmetic instruction case.

### 9.3.3 Store Instruction
Statement to be executed:
\[ M[r1] = r2 \]
Input Edge Logical Assertion:
\[ LA_{in} = (P_{in}, M_{in}) \]
Output Edge Extended State:
\[ \Sigma_{out} = (e_{out}, \sigma_{in}[\sigma_{in}(r2)/\sigma_{in}(r1)]) \]
Output Edge Logical Assertion:
\[ LA_{out} = (P_{in}, M_{in}[P_{in}, M_{in}(r2)/M_{in}(r1)]) \]
Program Generated from Output Edge \(LA_{out}\):
\[ f(LA_{out}, \sigma_{out}) = f(M_{in}[M_{in}(r2)/M_{in}(r1)], \sigma_{out}) \]
Executing generated program \(f(LA_{out}, \sigma_{out})\), we get the following state:
\[ \sigma'_{out} = \text{execute}(\sigma_0, f(M_{in}[M_{in}(r2)/M_{in}(r1)], \sigma_{out})) \]
Using the property:
\[ \sigma'_{out} = \text{execute}(\text{execute}(\sigma_0, f(M_{in}, \sigma_{out})), f(M[M_{in}(r1)] == M_{in}(r2), \sigma_{out})). \]
Since no loop is entered or exited with this instruction, \(\sigma_{in}\) and \(\sigma_{out}\) contain the same set of free variables. As a result,
\[ \sigma'_{out} = \text{execute}(\text{execute}(\sigma_0, f(M_{in}, \sigma_{in})), f(M[M_{in}(r1)] == M_{in}(r2), \sigma_{in})). \]
Note that, from the inductive hypothesis:
\[ \sigma_{in} = \text{execute}(\sigma_0, f(M_{in}, \sigma_{in})). \]
Therefore,
\[ \sigma'_{out} = \text{execute}(\sigma_{in}, f(M[M_{in}(r1)] == M_{in}(r2), \sigma_{in})). \]
Then, we get:
\[ \sigma'_{out} = \sigma_{in}[\sigma_{in}(r2)/\sigma_{in}(r1)] = \sigma_{out}. \]
Proof for the evaluation of the predicate at the outgoing edge is same as the arithmetic instruction case.

### 9.3.4 Two-way Branch
Statement to be executed:
\[ \text{if}(\text{cond}) \text{ then follow } e_{out1} \text{ else follow } e_{out2} \]
Input Edge Logical Assertion:
\[ LA_{in} = (P_{in}, M_{in}) \]
Output Edge Extended State:
\[ \Sigma_{out} = (e_{out}, \sigma_{out}), \]
where,
\[ e_{out} = \text{if}(\sigma_{in}(\text{cond})) \text{ then } e_{out1} \text{ else } e_{out2}, \]
\[ \sigma_{out} = \sigma_{in}. \]
Output Edge Logical Assertions on \(e_1\) and \(e_2\):
\[ LA_{out1} = (P_{out1}, M_{in}) \]
\[ LA_{out2} = (P_{out2}, M_{in}) \]
with
\[ P_{out1} = \text{simulate}(P_{in} \land M_{in}(\text{cond})) \]
\[ P_{out2} = \text{simulate}(P_{in} \land \neg M_{in}(\text{cond})) \]
Programs Generated from $LA_{out1}$ and $LA_{out2}$:
\[
\begin{align*}
  f(LA_{out1}, \sigma_{out}) &= f(M_{in}, \sigma_{out}) \\
  f(LA_{out2}, \sigma_{out}) &= f(M_{in}, \sigma_{out})
\end{align*}
\]
So, the programs generated from $LA_{out1}$ and $LA_{out2}$ are the same. The final state obtained from executing them:
\[
\sigma'_{out} = \text{execute} P(\sigma_0, f(M_{in}, \sigma_{out}))
\]
Since no loop is entered or exited with this branch instruction, $\sigma_{in}$ and $\sigma_{out}$ contain the same set of free variables. As a result,
\[
\sigma'_{out} = \text{execute} P(\sigma_0, f(M_{in}, \sigma_{in})) = \sigma_{in} = \sigma_{out}
\]
In order to prove the second part of the claim, we need to show:
\[
\begin{align*}
  \text{eval}(P_{out1}, \sigma_{out}) &= \text{true}, \\
  \text{eval}(P_{out2}, \sigma_{out}) &= \text{true}.
\end{align*}
\]
From the inductive hypothesis, we have:
\[
\begin{align*}
  \text{eval}(P_{in}, \sigma_{in}) &= \text{true}.
\end{align*}
\]
Considering the Boolean expression $\sigma_{in}(\text{cond})$, we have two cases:
1. If $\sigma_{in}(\text{cond}) = \text{true}$ then $\text{eval}(M_{in}(\text{cond}), \sigma_{in}) = \text{true}$. Combining this with the inductive hypothesis gives:
\[
\begin{align*}
  \text{eval}(P_{out1}, \sigma_{out}) &= \text{eval}(P_{in} \land M_{in}(\text{cond}), \sigma_{in}) \\
  &= \text{true}.
\end{align*}
\]
2. If $\sigma_{in}(\text{cond}) = \text{false}$ then $\text{eval}(\neg M_{in}(\text{cond}), \sigma_{in}) = \text{true}$. Combining this with the inductive hypothesis gives:
\[
\begin{align*}
  \text{eval}(P_{out2}, \sigma_{out}) &= \text{eval}(P_{in} \land \neg M_{in}(\text{cond}), \sigma_{in}) \\
  &= \text{true}.
\end{align*}
\]
Note that case (1) is realized when $e_{out} = e_{out1}$ and case (2) is realized when $e_{out} = e_{out2}$.

### 9.3.5 Two-way Join

Statement to be executed:
\[
\text{join}(e_1, e_2)
\]

Input Edge Logical Assertions:
\[
\begin{align*}
  LA_{in1} &= (P_{in1}, M_{in1}) \\
  LA_{in2} &= (P_{in2}, M_{in2})
\end{align*}
\]

Output Edge Extended State:
\[
\Sigma_{out} = (e_{out}, \sigma_{out}),
\]
where,
\[
\sigma_{out} = \sigma_{in}.
\]

Output Edge Logical Assertion:
\[
LA_{out} = (P_{out}, M_{out}),
\]

with the following equalities:
\[
\begin{align*}
  P_{out} &= \text{simplify}(P_{in1} \lor P_{in2}) \\
  M_{out} &= \text{simplify}(P_{out}, (P_{in1} \land M_{in1}) \lor (P_{in2} \land M_{in2}))
\end{align*}
\]

From inductive hypothesis, for the two incoming edges:
\[
\begin{align*}
  \text{execute} P(\sigma_0, f(M_{in1}, \sigma_{in})) &= \sigma_{in} \quad \text{, if entered from } e_{in1} \\
  \text{execute} P(\sigma_0, f(M_{in2}, \sigma_{in})) &= \sigma_{in} \quad \text{, if entered from } e_{in2}
\end{align*}
\]

Program Generated from $LA_{out}$:
\[
\begin{align*}
  f(LA_{out}, \sigma_{out}) &= f(M_{out}, \sigma_{out}) \\
  &= f(M_{in}, \sigma_{in}) \\
  &= f(\text{simplify}(P_{out}, (P_{in1} \land M_{in1}) \lor (P_{in2} \land M_{in2})), \sigma_{in})
\end{align*}
\]

Note that, $P_1 \land P_2 = \text{false}$, which means that the incoming edge predicates are mutually exclusive. Also note that from the inductive hypothesis:
\[
\begin{align*}
  \text{eval}(P_{in1}, \sigma_{in}) &= \text{true} \quad \text{, if entered from } e_{in1} \\
  \text{eval}(P_{in2}, \sigma_{in}) &= \text{true} \quad \text{, if entered from } e_{in2}
\end{align*}
\]
Then, if we do not perform any optimization/simplification, this program is equivalent to:

\[ f(LA_{out}, \sigma_{out}) \equiv \text{if } (\sigma_{in}(P_{in1})) \text{ then } f(M_{in1}, \sigma_{in}) \text{ else } f(M_{in2}, \sigma_{in}) \]

Executing the generated program gives the final state:

\[
\sigma'_{out} = \text{execute}P(\sigma_0, f(M_{out}, \sigma_{in}))
= \text{execute}P(\sigma_0, \text{if } (\sigma_{in}(P_{in1})) \text{ then } f(M_{in1}, \sigma_{in}) \text{ else } f(M_{in2}, \sigma_{in}))
= \text{if } (\sigma_{in}(P_{in1})) \text{ then } \text{execute}P(\sigma_0, f(M_{in1}, \sigma_{in})) \text{ else } \text{execute}P(\sigma_0, f(M_{in2}, \sigma_{in}))
= \text{if } (\sigma_{in}(P_{in1})) \text{ then } \sigma_{in} \text{ else } \sigma_{in} = \sigma_{out}.
\]

For the proof of the second part of the claim, we need to show:

\[ \text{eval}(P_{out}, \sigma_{out}) = \text{true}, \]

The inductive hypothesis states that for the two incoming edges,

\[
\text{eval}(P_{in1}, \sigma_{in}) = \text{true} \quad \text{if entered from } e_{in1} \\
\text{eval}(P_{in2}, \sigma_{in}) = \text{true} \quad \text{if entered from } e_{in2}
\]

From the inductive hypothesis and \( \sigma_{in} = \sigma_{out} \), we get:

\[
\text{eval}(P_{in1}, \sigma_{out}) = \text{true} \quad \text{if entered from } e_{in1} \\
\text{eval}(P_{in2}, \sigma_{out}) = \text{true} \quad \text{if entered from } e_{in2}
\]

Combining these two statements with a logical or operator gives us (independent of which edge this node is entered):

\[
\text{eval}(P_{out}, \sigma_{out}) = \text{eval}(P_{in1} \lor P_{in2}, \sigma_{out}) = \text{true}.
\]

### 9.3.6 Loop Entry Join Node

Statement to be executed:

\[ \text{LoopEntryJoinNode}(I) \]

Input Edge Logical Assertions on edge \( e_{in1} \) and \( e_{in2} \):

\[
LA_{in1} = (P_{in1}, M_{in1}) \\
LA_{in2} = (P_{in2}, M_{in2})
\]

Output Edge Extended State:

\[ \Sigma_{out} = (e_{out}, \sigma_{out}), \]

where,

\[ \sigma_{out} = \sigma_{in}. \]

Output Edge Logical Assertion:

\[ LA_{out} = (P_{out}, M_{out}), \]

From the definition of loop entry join node, the following equality on the predicate holds:

\[ P_{out} = P_{in1} \land (0 \leq I < I_{\text{FINISH}}() - 1) \]

The output expression map:

\[ M_{out} = \{ \text{delete}(M_{in1}, M[I] = 0) ; \text{for}(0 \leq j \leq I - 1) [ LB(j) ] \} \]

Program Generated from the logical assertion \( LA_{out} \) on outgoing edge:

\[ f(LA_{out}, \sigma_{out}) = f(M_{out}, \sigma_{out}) \]

Executing this modified program, we get:

\[
\sigma'_{out} = \text{execute}P(\sigma_0, f(M_{out}, \sigma_{out}))
= \text{execute}P(\sigma_0, f(\text{delete}(M_{in1}, M[I] = 0) ; \text{for}(0 \leq j \leq I - 1) [ LB(j) ], \sigma_{out}))
= \text{execute}P(\sigma_0, f(M_{in1}[\sigma_{out}(I)/I] ; \text{for}(0 \leq j \leq I - 1) [ LB(j) ], \sigma_{out})),
\]

since all uses of \( I \) uses the value obtained from \( \sigma_{out} \). Note that this node can be entered either through the forward edge or through the back edge.
1. If this node is entered through the forward edge, then \(\sigma_{out}(I) = 0\), therefore,
\[
\begin{align*}
\sigma'_{out} &= \text{executeP}(\sigma_0, f(M_{in1}[0/I]; for(0 \leq j \leq -1)[LB(j)],[\sigma_{out}])) \\
&= \text{executeP}(\sigma_0, f(M_{in1})) \\
&= \text{executeP}(\sigma_0, f(M_{in1}, \sigma_{in})) \\
&= \sigma_{in}
\end{align*}
\]

2. If this node is entered through the back edge, then \(\sigma_{out}(I) = \sigma_{in}(I)\), therefore,
\[
\begin{align*}
\sigma'_{out} &= \text{executeP}(\sigma_0, f(M_{in2}[\sigma_{in}(I)/I]; for(0 \leq j \leq I - 1)[LB(j)],[\sigma_{in}])) \\
&= \text{executeP}(\sigma_0, f(M_{in2}, \sigma_{in})) \\
&= \sigma_{in}
\end{align*}
\]

Note that we have the following equality:
\[
M_{in2} = \text{delete}(M_{in1}, M[I] == 0); for(0 \leq j \leq I - 1)[LB(j)]
\]
Which gives,
\[
f(M_{in2}, \sigma_{in}) \equiv f(M_{in1}[\sigma_{in}(I)/I]; for(0 \leq j \leq I - 1)[LB(j)],[\sigma_{in}])
\]
Therefore, we obtain,
\[
\begin{align*}
\sigma'_{out} &= \text{executeP}(\sigma_0, f(M_{in2}, \sigma_{in})) \\
&= \sigma_{in}
\end{align*}
\]

Combining the two cases gives us the following,
\[
\sigma'_{out} = \sigma_{in} = \sigma_{out}.
\]

For the second part of the proof, the inductive hypothesis states that:
\[
\text{eval}(P_{in}, \sigma_{in}) = \text{true}.
\]
We need to show the following:
\[
\text{eval}(P_{out}, \sigma_{out}) = \text{true}.
\]
Starting from \(P_{out}\) and using \(\sigma_{out} = \sigma_{in}\), we get:
\[
\text{eval}(P_{in} \land (0 \leq I \leq I_{\text{FINISH}}() - 1), \sigma_{in}).
\]
Note, however, that from the definition of \(I\), \(\sigma_{in}(I)\) always satisfies the condition: \(0 \leq \sigma(I) \leq I_{\text{FINISH}}() - 1\). Therefore, also using the inductive hypothesis, we get:
\[
\begin{align*}
\text{eval}(P_{out}, \sigma_{out}) &= \text{eval}(P_{in} \land (0 \leq I \leq I_{\text{FINISH}}() - 1), \sigma_{in}) \\
&= \text{true}.
\end{align*}
\]

9.3.7 Loop Exit Node

Statement to be executed:

\(\text{LoopExitNode}(I)\)

Input Edge Logical Assertion:

\(LA_{in} = (P_{in}, M_{in})\)

Output Edge Extended State:

\(\Sigma_{out} = (e_{out}, \sigma_{out})\)

Output Edge Logical Assertion:

\(LA_{out} = (P_{out}, M_{out})\),

where,
\[
\begin{align*}
P_{out} &= \text{delete}(P_{in}, 0 \leq I \leq I_{\text{FINISH}}() - 1) \\
M_{out} &= \text{replace}(M_{in}, I_{\text{FINISH}}() - 1, I) \land M[I] == -1)
\end{align*}
\]

Code generated from logical assertion:
\[
f(M_{out}, \sigma_{out}) = f(\text{replace}(M_{in}, I_{\text{FINISH}}() - 1, I) \land M[I] == -1, \sigma_{out}),
\]
where,
\[
I_{\text{FINISH}}() - 1 = \min\{I \mid I \geq 0 \land \text{cond}(I) == \text{false}\}.
\]
Note that on edge $e_{out}$, the loop is exited. Considering the whole loop as a virtual box, the output state is expected to be equivalent to execution of this loop $\sigma_{in}(I)$ times. From the definition of loop index variable $I$, it is the smallest number that satisfies the loop exit condition:

$$\sigma_{in}(I) = \min\{I \mid I \geq 0 \land \text{cond}(I) == \text{false}\} = I_{FINISH}() - 1.$$ 

Therefore, $f(M_{out}, \sigma_{out})$ is equivalent to:

$$f(M_{out}, \sigma_{out}) \equiv f(\text{replace}(M_{in}, \sigma_{in}(I), I) \land M[I] == -1, \sigma_{out}).$$ 

Executing this program gives:

$$\sigma'_{out} = \text{executeP}(\sigma_0, f(\text{replace}(M_{in}, \sigma_{in}(I), I) \land M[I] == -1, \sigma_{out})).$$

Since all uses of $I$ is replaced to obtain $M_{out}$, we can convert the connector into a semicolon and apply the property:

$$\sigma'_{out} = \text{executeP}(\text{executeP}(\sigma_0, f(\text{replace}(M_{in}, \sigma_{in}(I), I), \sigma_{out})), f(M[I] == -1, \sigma_{out})).$$

Similarly, since these logical assertions do not contain any free occurrences of $I$, we can replace $\sigma_{out}$ with $\sigma_{in}$ inside the function $f$:

$$\sigma'_{out} = \text{executeP}(\text{executeP}(\sigma_0, f(\text{replace}(M_{in}, \sigma_{in}(I), I), \sigma_{in})), f(M[I] == -1, \sigma_{in})).$$

From inductive hypothesis, we already have:

$$\sigma_{in} = \text{executeP}(\sigma_0, f(M_{in}, \sigma_{in})).$$

From this hypothesis, we already know the identity:

$$\sigma_{in}(I) = \text{executeP}(\sigma_0, f(M_{in}, \sigma_{in}))(I).$$

This means that executing the program obtained from $M_{in}$ already results with a value of $I$ equal to $\sigma_{in}(I)$. This information eliminates the replace operation in the equation above and gives:

$$\sigma'_{out} = \text{executeP}(\text{executeP}(\sigma_0, f(M_{in}, \sigma_{in})), f(\{M[I] == -1\}, \sigma_{in})) = \text{executeP}(\sigma_0, f(M[I] == -1, \sigma_{in})) = \sigma_{in}[-1/I] = \sigma_{out}.$$

For the second part of the proof, from inductive hypothesis, we have:

$$\text{eval}(P_{in}, \sigma_{in}) = \text{true}.$$

We are trying to show that:

$$\text{eval}(P_{out}, \sigma_{out}) = \text{true}.$$

From the symbolic execution result for $P_{out}$ and the equality $\sigma_{out} = \sigma_{in}[-1/I]$, we have:

$$\text{eval}(P_{out}, \sigma_{out}) = \text{eval}(\text{delete}(P_{in}, 0 \leq I \leq I_{FINISH}() - 1), \sigma_{in}[-1/I]) = \text{eval}(\text{delete}(P_{in}, 0 \leq I \leq I_{FINISH}() - 1), \sigma_{in}),$$

as the predicate no longer contains $I$. Note that deleting a condition from a predicate is equivalent to reducing the strength of the conditions that must be satisfied, which gives the following implication:

$$\text{eval}(P_{in}, \sigma_{in}) = \text{true} \Rightarrow \text{eval}(\text{delete}(P_{in}, 0 \leq I \leq I_{FINISH}() - 1), \sigma_{in}) = \text{true}.$$

Therefore, the inductive hypothesis already implies that this condition is satisfied.

### 9.3.8 Loop Entry Node

Statement to be executed:

$$\text{LoopEntryNode}(I)$$

Input Edge Logical Assertion:

$$LA_{in} = (P_{in}, M_{in})$$

Output Edge Extended State:

$$\Sigma_{out} = (e_{out}, \sigma_{in}[0/I])$$

Output Edge Logical Assertion:

$$LA_{out} = (P_{in}, M_{in}[P_{in}, 0/I])$$

Program Generated from Output Edge $LA_{out}$:

$$f(LA_{out}, \sigma_{out}) = f(M_{in}[0/I], \sigma_{out})$$

Executing generated program $f(LA_{out}, \sigma_{out})$, we get the following state:

$$\sigma'_{out} = \text{executeP}(\sigma_0, f(M_{in}[0/I], \sigma_{out}))$$
Note that there is no free occurrence of variable $I$ in $M_{in}[0/I]$, therefore
\[
\sigma_{out}' = \text{execute}P(\sigma_0, f(M_{in}[0/I], \sigma_{in}) \\
= \sigma_{in}[0/I] \\
= \sigma_{out}.
\]
For the second part of the proof, since $P_{in} = P_{out}$ and $\sigma_{in}$ and $\sigma_{out}$ differ only in the variable $I$ which does not exist in $P_{in}$ or $P_{out}$, the condition to show is trivially equivalent to the inductive hypothesis.

### 9.3.9 Loop Exit Branch Node
Similar to standard two-way branch except that predicate is not extended with $\text{cond}$ on the forward edge. Statement to be executed:
\[
\text{if}(\text{cond}) \text{ then follow } e_{out1} \text{ else follow } e_{out2}
\]

Input Edge Logical Assertion:
\[
LA_{in} = (P_{in}, M_{in})
\]

Output Edge Extended State:
\[
\Sigma_{out} = (e_{out}, \sigma_{out}),
\]
where,
\[
e_{out} = \text{if}(\sigma_{in}(\text{cond})) \text{ then } e_{out1} \text{ else } e_{out2} \\
\sigma_{out} = \sigma_{in}.
\]

Output Edge Logical Assertions on $e_{out1}$ and $e_{out2}$:
\[
LA_{out1} = (P_{in}, M_{in}) \\
LA_{out2} = (P_{in} \land \neg\text{cond}(I), M_{in})
\]

Programs Generated from $LA_{out1}$ and $LA_{out2}$:
\[
f(LA_{out1}, \sigma_{out}) = f(M_{in}, \sigma_{out}) \\
f(LA_{out2}, \sigma_{out}) = f(M_{in}, \sigma_{out})
\]

So, the programs generated from $LA_{out1}$ and $LA_{out2}$ are the same. The final state obtained from executing them:
\[
\sigma_{out}' = \text{execute}P(\sigma_0, f(M_{in}, \sigma_{out}))
\]

Since no loop is entered or exited with this branch instruction:
\[
\sigma_{out}' = \text{execute}P(\sigma_0, f(M_{in}, \sigma_{in})) = \sigma_{in} \\
= \sigma_{out}.
\]
For the second part of the proof, on the forward edge, since $P_{out} = P_{in}$ and $\sigma_{out} = \sigma_{in}$, the claim is the same as the inductive hypothesis and trivially holds. For the back edge, $P_{out} = P_{in} \land \neg\text{cond}(I)$ and $\sigma_{out} = \sigma_{in}$. Since the program follows the back edge, we know that $\sigma_{in}(\text{cond}(I)) = \sigma_{out}(\text{cond}(I)) = \text{false}$. Therefore,
\[
\text{eval}(P_{in} \land \neg\text{cond}(I), \sigma_{out}) = \text{true}.
\]

### 9.3.10 Loop Back Edge Node
Statement to be executed:
\[
\text{LoopBackEdgeNode}(I)
\]

Input Edge Logical Assertion:
\[
LA_{in} = (P_{in}, M_{in})
\]

Output Edge Extended State:
\[
\Sigma_{out} = (e_{out}, \sigma_{out}) = (e_{out}, \sigma_{in}[\sigma_{in}(I) + 1/I])
\]

Output Edge Logical Assertion:
\[
LA_{out} = (P_{out}, M_{out}),
\]
where,
\[
P_{out} = P_{in} \\
M_{out} = M_{in}[\text{simplify}(P_{in}, M_{in}(I) - 1)/I].
\]

The program generated from $LA_{out}$:
\[
f(LA_{out}, \sigma_{out}) = f(M_{in}[\text{simplify}(P_{in}, M_{in}(I) - 1)/I], \sigma_{out}) \\
= f(M_{in}[\text{simplify}(P_{in}, M_{in}(I) - 1)/I], \sigma_{in}[\sigma_{in}(I) + 1/I])
\]
This means that we replace all $I$s with $(I - 1)$s, but increment the value of $I$. The number of loop iterations executed remains constant, but the value of $I$ is incremented. Executing this program gives:

$$
\sigma'_\text{out} = \text{execute}P(\sigma_0, f(M_{in}[\text{simplify}(P_{in}, M_{in}(I) - 1/I], \sigma_{in}[\sigma_{in}(I) + 1/I]))
\text{execute}P(\text{execute}P(\sigma_0, f(M_{in}, \sigma_{in}), \{I = I + 1\})
\text{execute}P(\sigma_{in}, \{I = I + 1\})
\sigma_{in}[\sigma_{in}(I) + 1/I]
\sigma_{out}.
$$

For the second part of the proof, we have $P_{out} = P_{in}$. Note that $P_{in}$ already contained $0 \leq I \leq I_{\text{FINISH}} - 1 \land \neg \text{cond}(I)$ as the condition over $I$. From the inductive hypothesis, we have:

$$\text{eval}(0 \leq I \leq I_{\text{FINISH}} - 1 \land \neg \text{cond}(I), \sigma_{in}) = \text{true}.$$ 

For the outgoing edge, using $\sigma_{out} = \sigma_{in}[(\sigma_{in}(I) + 1)/I]$, we get:

$$\text{eval}(0 \leq I \leq I_{\text{FINISH}} - 1, \sigma_{out}) = \text{eval}(0 \leq I + 1 \leq I_{\text{FINISH}} - 1, \sigma_{in})$$

Note that by definition of $I_{\text{FINISH}} - 1$, it is the minimum $I$ value that satisfies $\text{cond}(I)$. As the inductive hypothesis also states that $\neg \text{cond}(I) = \text{true}$, we know that $I < I_{\text{FINISH}} - 1$. Therefore, $I + 1 \leq I_{\text{FINISH}} - 1$, which gives

$$\text{eval}(0 \leq I \leq I_{\text{FINISH}} - 1, \sigma_{out}) = \text{eval}(0 \leq I + 1 \leq I_{\text{FINISH}} - 1, \sigma_{in}) = \text{true}. $$